Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Evaluation von Strategien für die Konstruktion von Contraction Hierarchies

Aimn Ahmed

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Tobias Rupp
Beginn am:	2. Januar 2020
Beendet am:	27. August 2020

Kurzfassung

Eine altbewährte Lösung des shortest-path-problem liefert der Dijkstra-Algorithmus. In den vergangenen 20 Jahren wurden dennoch diverse Techniken zur Verbesserung der Effizienz entwickelt. Eine dieser Techniken stellen die Contraction Hierarchies dar, welche auf einer Vorverarbeitung des Graphen basieren. Auf sehr dichten Graphen, beispielsweise bei realen Straßennetzwerken, stellte sich der Dijkstra als ineffizient heraus. Mithilfe dieser Voerverarbeitungstechnik Contraction Hierarchie kann eine Suchanfrage um ein Vielfaches schneller beantwortet. Eine weitere bekannte Technik, zur Verschnellerung solcher Anfragen, stellen Hub-Labels dar. Es kann dadurch ein Distanzorakel aufgebaut werden, sodass die Antwortzeit auf eine kürzeste Distanzanfrage im Bereich weniger Mikrosekunden liegt. Die vorliegende Bachelorarbeit stellt verschiedene Strategien zur Erstellung einer Contraction Hierarchie vor und bemisst ihre Qualität an der Menge der berechneten Hub-Labels. In einem experimentellen Vergleich, wurden verschiedene Strategien vorgestellt, evaluiert und miteinander verglichen. Dabei wurde der "Tradeoff"zwischen der Berechnungszeit und der gemessenen Qualität ebenfalls berücksichtigt.

Inhaltsverzeichnis

1 Einleitung **13**

2 Grundlagen **15**
2.1 Graphentheorie 15
2.2 Dijkstra-Algorithmus 15
2.3 Contraction Hierarchies 16

3 Heuristiken zur Erstellung von Conctraction Hierarchies **25**
3.1 Problemstellung 25
3.2 Hub-Score 25
3.3 Dijkstra-Score 29
3.4 Heuristiken 29

4 Messungen und Evalutaion **33**
4.1 Testgraphen und Durchführung 33

5 Zusammenfassung und Ausblick **37**

Literaturverzeichnis **39**

Abbildungsverzeichnis

2.1 Kontraktionsbeispiele 19

3.1 Screenshot aus Visualisier: Die blauen Rechtecke stehen für die hubs der Startknoten und die orangen Rechtecke stehen für die hubs der Zielknoten. Die lilalen Rehctecken entsprechen den gemeinsamen hubs. 26

3.2 Hub-Scoreverteilung 28

Tabellenverzeichnis

4.1 Testgraphen - BR = bremen, LM = lower Manhattan, MV = Mecklenburg-Vorpommern 33
4.2 OBO 34
4.3 QC 34
4.4 ED 34
4.5 SC 34
4.6 SCED 35
4.7 LM-CH-ED-SC: Score-Heurstik wurde auf den LM-CH-Graph, der mithilfe der ED bestimmt wurde, angewendet. HS-E = Hub-Score nach Anzahl Kante auf OTA-Dijkstra; HS-E-Up = analag für OTA-CH-Dijkstra 35
4.8 LM-CH-ED-SCED: Score-And-Edge-Difference-Heurstik wurde auf den LM-CH-Graph, der mithilfe der ED bestimmt wurde, angewendet. 35

Verzeichnis der Algorithmen

2.1 Dijkstra-Algorithmus 17
2.2 CH-Konstruktion 18
2.3 Knotenkontraktion 18
2.4 Hub-Label-Berechnung 22
2.5 Kürzen der Hubs 22

[T1]fontenc

1 Einleitung

Der Dijkstra-Algorithmus ist heute kaum noch wegzudenken. Ein triviales Vorgehen, bei dem ein kürzester Weg von A nach B ermittelt wird. Er gilt als einer der fundamentalen Algorithmen zur Bestimmung kürzester Wege. Um solch eine Anfrage zu lösen, benötigt der Algorithmus die geeigneten Informationen, um von einem Standort zum nächsten zu gelangen. Da Karten von Straßennetzwerken oft komplex und überladen sind mit unnötigen Informationen, die zur Lösung dieser Frage nicht hilfreich sind, wäre es wünschenswert, solche Straßennetzwerke auf eine geeignete übersichtliche Weise zur mathematischen Analyse abzubilden. Eine abstrakte Struktur, die zum einen alle nötigen Informationen zur Lösung dieses Problems enthält und zum anderen alle unnützen Details eines Straßennetzwerks dabei ausblendet, wären Graphen. Ein Graph eignet sich hervorragend zur Berechnung kürzester Pfade, da zum einen alle Positionen, alle Verbindungen zwischen diesen Positionen, sowie alle entstehenden Reisekosten simpel dargestellt werden können.

Obwohl in den letzten 20 Jahren das shortest-path-problem als verstanden und gelöst gilt, gab es eine enorme Entwicklung bezüglich der Effizienz bei der Lösung einer Suchanfrage. Zwei Techniken zur Beschleunigung stellen die Contraction Hierarchies (CH) und das darauf basierende Hub-Labelling dar. Diese werden im Laufe dieser Arbeit im Wesentlichen untersucht. Das Augmentieren von Graphen mithilfe der Contraction Hierarchies beschleunigt das Lösen eines gegebenen shortest-path-problems um ein Vielfaches. Dabei werden in sogenannten Kontraktionsrunden bestimmte Teile des Graphen entfernt. Bei der Kontraktion eines Knotens, wird er aus dem Graphen entfernt. Des Weiteren werden all seine Kanten durch sogenannte *Shortcuts* ersetzt. Nachdem jeder Knoten kontrahiert, erhält man einen neuen Graphen mit zusätzlichen Shortcuts. Während der Kontraktionsrunden werden jedem Knoten ein Level zugewiesen. Die Dijkstra-Suche findet nun bidirektional statt, wobei nur Kanten betrachtet werden, die zu einem Knoten mit einem höheren Level führen. Mithilfe dieser Shortcuts und Level werden große Teile des Graphen übersprungen und resultieren so in kleineren Suchbäumen, die trotzdem immer eine korrekte Lösung liefern. Bei kürzesten Distanzanfrage werden dennoch Teile des Graphen exploriert, wodurch unnötiger Aufwand betrieben wird, da man lediglich die Distanz haben möchte. Für dieses Problem gibt es eine Technik namens Hub-Label. Indem man für jeden Knoten eine hinreichend kleine Liste berechnet, die eine Menge von Knoten und die dazugehörige kürzeste Distanz zu ihnen hält, kann eine kürzeste Distanzanfrage in Mikrosekunden beantwortet werden. Die Berechnung dieser Listen hängt dabei stark von Kontraktionsreihenfolge der Contraction Hierarchy ab. Untersucht werden nun Strategien, welche eine unterschiedliche Kontraktionsreihenfolge festlegen. Die dadurch entstandenen Listen bei der Berechnung der Hub-Labels liefern eine Basis für den Vergleich dieser Heuristiken.

Gliederung

Kapitel 2 - Grundlagen: In diesem Kapitel werden die nötigen Grundlagen zur Graphentheorie und Contraction Hierarchies eingeführt und formal definiert.

Kapitel 3 - Heuristiken zu Erstellung von Contraction Hierarchies: Dieses Kapitel beschreibt die Problemstellung, sowie die verwendeten Heuristiken, um diese Problemstellung zu bewältigen.

Kapitel 4 - Messungen und Evaluation: Hier werden alle durchgeführten Tests, sowie die zugehörige Evaluation der Ergebnisse dokumentiert.

Kapitel 5 - Zusammenfassung und Ausblick: Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen und liefert einen Ausblick auf mögliche zukünftige Arbeit in diesem Fachgebiet.

2 Grundlagen

2.1 Graphentheorie

Wie in Kapitel 1 bereits erwähnt, lässt sich ein reales Straßennetzwerk abstrahieren und übersichtlich als Graph darstellen. Ein Graph sei dabei definiert als ein Tupel $G(V, E)$, wobei V die endliche Knotenmenge und $E \subseteq V \times V$ die Kantenmenge darstellt. Ein weiterer wichtiger Aspekt sind die Kosten beim Durchlaufen einer Kante. Sei $cost : E \longrightarrow \mathbb{R}^+$ die Kostenfunktion, die jeder Kante ihre Kosten zuweist. Wichtig hierbei ist, dass nur positive Kosten berücksichtigt werden, da der Dijkstra-Algorithmus dies als Bedingung voraussetzt. Diese Graphen nennt man positiv gewichtete Graphen. Jede Kante $e = (u, v) \in E$ in einem gerichteten Graph ist ein geordnetes Paar von Knoten, bei dem der Startknoten $source(e) = u$, der Zielknoten $target(e) = v$ und $cost(e)$ die Kosten von u nach v darstellt. Existiert für jede Kante $(u, v) \in E$ eine Kante $(v, u) \in E$ mit $cost((u, v)) = cost((v, u))$, so spricht man von einem ungerichteten Graphen. In einem ungerichteten Graph sind Kanten ungeordnete Knotenpaare. Zwei Knoten $u, v \in V$ werden als adjazent bezeichnet, wenn eine Kante $(u, v) \in E$ existiert.

Sei nun ein Pfad p der Länge k von einem Startknoten s zu einem Zielknoten t definiert als eine Abfolge von Knoten $p_{s,t} = v_0, v_2, ..., v_{k-1}$ bei dem $\forall i \in 0, 1, ..., k-1 : v_i \in V$ und $s = v_0, t = v_{k-1}$ mit $\forall v_i \in p_{s,t} : (v_i, v_{i+1}) \in E$. Aufgrund dieser Definition lassen sich Pfade alternativ auch als eine Abfolge von Kanten wie folgt definieren: $\hat{p}_{s,t} = e_0, e_1, ..., e_{k-2} = (v_0, v_1), (v_1, v_2), ..., (v_{k-2}, v_{k-1})$. Die Kosten eines Pfades von s nach t mit Länge k können nun mithilfe der Kostenfunktion direkt über die Summe aller Kosten der einzelnen Kanten des Pfades ermittelt werden: $cost(p) = cost(\hat{p}) = \sum_{i=0}^{k-2} cost(e_i)$.

Als $d(u, v)$ bezeichnet man die minimalen Kosten eines Pfades von u nach v. Existiert kein Pfad von u nach v so gilt $d(u, v) = \infty$. Folglich wird ein Pfad $p_{u,v} = e_0, e_1, ..., e_{k-2}$ als kürzester Pfad von u nach v bezeichnet, wenn $cost(p_{u,v}) = cost(\hat{p}_{u,v}) = \sum_{i=0}^{k-2} cost(e_i) = d(u, v)$.

2.2 Dijkstra-Algorithmus

Die Ermittlung des kürzesten Pfades in einem positiv gewichteten Graphen $G(V, E)$ von einem Startknoten u zu einem Zielknoten v mit $u, v \in V$ kann über diverse Suchalgorithmen geschehen. Einer dieser Suchalgorithmen ist der Dijkstra-Algorithmus [Dij59]. Neben dem

gegebenen Graphen $G(V, E)$ werden alle weiteren nötigen Informationen für eine effiziente Berechnung des kürzesten Pfades von einem Startknoten s zu einem Zielknoten t in drei Mengen gespeichert: $PQ, d, predecessor$. In der Menge d werden für alle Knoten $v \in V$ die bisherigen Pfadkosten $cost(p_{s,v})$ gespeichert. Diese können sich während des Algorithmus mehrmals ändern, da es mehrere Pfade von s nach v geben kann. Dementsprechend gilt für den Startknoten s initial $d[s] = 0$, während für alle anderen Knoten $v \in V \setminus s$ der Wert $d[v] = \infty$ gespeichert wird. Zusätzlich zu den Kosten werden auch die zugehörigen Pfade abgespeichert. Dafür wird die Menge $predecessor$ verwendet. In $predecessor$ wird für jeden Knoten $v \in V$ der benachbarte Vorgängerknoten $u \in V$ gespeichert, mit dem v erreicht wurde: $predecessor[v] = u$. Auf diese Weise ist es später möglich, alle Pfade $p_{s,v}$ rückwärts von v nach s zu durchlaufen und den Pfad zu rekonstruieren. Initial sind alle Werte $predecessor[v] = null$. Die Menge PQ stellt eine Prioritätswarteschlange dar. Sie enthält Knoten $v \in V$ und deren Kosten $d[v]$ in aufsteigender Reihenfolge nach ihrem Kostenwert $d[v]$. Initial enthält PQ lediglich den Startknoten s.

Der Dijkstra-Algorithmus entfernt nun in jedem Schritt das vorderste Element $u \in PQ$ mit dem kleinsten $d[u]$ aus PQ und relaxiert alle Kanten $(u, v) \in E$. Beim Relaxieren einer Kante (u, v) wird geprüft, ob $(d[u] + cost((u, v))) < d[v]$ ist. Ist dies der Fall, so werden die Kosten $d[v] = d[u] + cost((u, v))$ und der Vorgängerknoten $predecessor[v] = u$ aktualisiert. Außerdem wird v mit seinen neuen Kosten $d[v]$ in die Prioritätswarteschlange PQ eingefügt und korrekt nach $d[v]$ einsortiert. In einer anderen Variante könnte man die Knoten jeweils einmal in PQ einfügen und ihre Kosten ändern, statt sie ein weiteres Mal hinzuzufügen. Im verwendeten Algorithmus 2.1 kann ein Knoten mehrfach mit verschiedene Kosten in PQ vorkommen. Sobald ein Knoten v aus der PQ entfernt wird, so gilt als v als $bearbeitet$. Ein Knoten v kann nur einmal $bearbeitet$ werden, da $\forall u \in PQ : d[u] >= d[v]$. Da der Dijkstra-Algorithmus einen positiv gewichteten Graphen voraussetzt und damit negativ gewichtete Kanten nicht existieren, ist es nicht mehr möglich den Knoten v über einen Knoten $u \in PQ$ mit einem kürzeren Pfad zu erreichen. Sobald der Zielknoten t aus PQ entfernt wurde, terminiert der Algorithmus. Mithilfe der zwei Mengen d und $predecessor$ können sowohl der kürzeste Pfad $p_{s,t}$, als auch die zugehörigen Kosten $d(s, t)$ des kürzesten Pfades angegeben werden. Sollte kein Pfad zwischen den Knoten s und t existieren, so terminiert der Algorithmus ebenfalls und liefert $null$ als Ergebnis zurück.

2.3 Contraction Hierarchies

Contraction Hierarchies [Del08] ist eine Vorverarbeitungstechnik, welche auf einem Graph angewendet werden kann. Sie hat das Ziel die Suche nach einem kürzesten Pfad im Graphen zu beschleunigen. Dazu wird der Graph in einer Vorverarbeitungsphase um sogenannte Abkürzungskanten (Shortcuts) erweitert. Shortcuts fassen dabei mehrere Kanten zu einer Kante zusammen. Des Weiteren wird jedem Knoten ein Level zugewiesen, wobei das Level die Relevanz eines Knotens widerspiegelt. Durch diese Erweiterungen werden bei einer Dijkstra-Anfrage deutlich weniger Kanten exploriert, wodurch eine Suchanfrage schneller beantwortet werden kann. Ein Graph auf dem die Vorverarbeitungstechnik Contraction Hierarchies (CH) angewendet wurde, nennt man CH-Graph.

Algorithmus 2.1 Dijkstra-Algorithmus

```
function DIJKSTRA(V,E,cost,s,t)
    PQ ← PriorityQueue()
    predecessor ← Array(V.size())
    d ← Array(V.size())

    for all v ∈ V do
        d[v] ← ∞
        predecessor[v] ← null
    end for
    d[s] ← 0

    PQ.push(s, d[s])
    while !PQ.isEmpty() do
        PQElement pq ← PQ.popMinCost()

        if pq.node == t then
            return d, predecessor                    // Ziel wurde erreicht
        end if

        if pq.cost > d[pq.node] then
            continue                 // bereits bearbeitete Knoten werden ignoriert
        end if

        for all Edge e = (u, v) ∈ E do
            if d[v] > (d[u] + cost[e]) then
                d[v] = d[u] + cost(e)
                predecessor[v] = u
                PQ.push(v, d[v])
            end if
        end for
    end while
    return null                      // es wurde kein Pfad von s nach t gefunden
end function
```

2.3.1 Vorverarbeitung

Contraction Hierarchies basieren, wie der Name bereits andeutet, auf der Kontraktion von Knoten. Hierbei wird eine bestimmte Reihenfolge festgelegt, in der die Knoten nacheinander kontrahiert werden. Die Reihenfolge spiegelt dabei die Relevanz des Knotens wider. Die Kontraktionsreihenfolge bestimmt gleichzeitig auch das zugewiesene Level der Knoten. Außerdem hängt das eigentliche Ziel der CH, die Beschleunigung von kürzesten Pfadanfragen, stark von dieser Kontraktionsreihenfolge ab.

Algorithmus 2.2 CH-Konstruktion

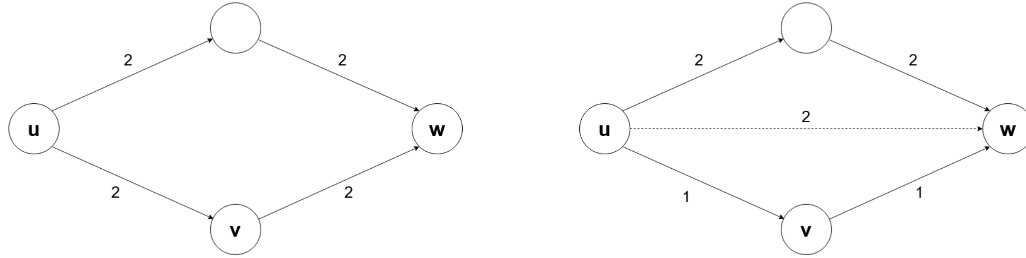
```
function chConstructor(V,E,cost,level,order)
    V.sortBy(order)                          // Kontraktionsreinfolge festlegen
    counter ← 0
    while V.size() > 1 do
        Node n ← V.top()                     // vorderster Knoten in V wird kontrahiert
        contract(V, E, cost, n)
        level(n) ← counter++
    end while
    return V, E, cost, level
end function
```

Algorithmus 2.3 Knotenkontraktion

```
function contract(V,E,cost,n)
    for all Edge e = (u, n) ∈ E do
        for all Edge e = (n, v) ∈ E do
            if (u, n, v) is the only shortest path from u to v then
                Edge shortcut ← (u, v)
                cost(u, v) ← cost(u, n) + cost(n, v)
                E ← add shortcut to E
            end if
        end for
    end for
    return V, E, cost
end function
```

Um aus einem Graphen G einen CH-Graphen zu erstellen, werden, wie im Algorithmus 2.2 beschrieben, schrittweise Knoten kontrahiert. Bei der Kontraktion (siehe Algorithmus 2.3) eines Knoten $v \in V$ wird der Knoten aus der Knotenmenge V entfernt. Dabei muss weiterhin gewährleistet werden, dass alle kürzesten Pfade, die über diesen Knoten v verlaufen, weiterhin erhalten bleiben. Alle Pfade, die durch die Kontraktion von v beeinträchtigt werden, müssen über den Knoten v verlaufen und enthalten damit die Kanten$(u, v) \in E$ und $(v, w) \in E$. Um diese Beeinträchtigung zu kompensieren, betrachtet man alle Knoten, die zu v adjazent sind. Alle Vorgängerknoten u mit $(u, v) \in E$ und Nachfolgerknoten w mit $(v, w) \in E$ werden paarweise überprüft. Dabei soll festgestellt werden, ob $p_{u,w} = u, v, w$ der einzige kürzeste Pfad von u nach w ist. Festgestellt wird dies, indem man eine kürzeste Pfadanfrage von u nach w startet und dabei den Knoten v ignoriert. Falls es keinen anderen Pfad gibt, der von u nach w führt, den Knoten v ignoriert und dessen Kosten nicht höher als $cost(u, v) + cost(v, w)$ sind, so wird eine Abkürzungskante (u, w) mit Kosten $cost(u, v) + cost(v, w)$ der Kantenmenge E hinzugefügt. Sollte jedoch ein anderer Pfad von u nach w existieren, der nicht über v läuft und dessen Kosten kleiner gleich $cost(u, v) + cost(v, w)$ sind, so wird lediglich der Knoten v aus der Knotenmenge V entfernt. Dadurch werden unnötige Abkürzungskanten ignoriert, die eine Dijkstra-Anfrage lediglich verlangsamt hätten. Dieses Vorgehen stellt daher sicher, dass sämtliche kürzeste Pfade im Graphen weiterhin erhalten bleiben. Außerdem wird bei der Kontraktion dem

Knoten ein Level zugewiesen. Die Levelfunktion sei dabei definiert als $l : V \to \mathbb{N}_0$. Eine Veranschaulichung liefern die zwei Abbildungen 2.1a und 2.1b, bei der jeweils ein Knoten kontrahiert wird.



(a) Nach der Kontraktion des Knotens v, wird dem Graphen kein neuer Shortcut hinzugefügt, da u,v,w nicht der einzige kürzeste Pfad von v nach w ist.

(b) Nach der Kontraktion des Knotens v, wird dem Graphen ein neuer Shortcut (u,w) hinzugefügt, da u,v,w der einzige kürzeste Pfad von v nach w ist.

Abbildung 2.1: In den gezeigten Abbildungen wird jeweils der Knoten v in dem gegebenen Graphen kontrahiert. Im Fall (a) entsteht dabei kein neuer Shortcut, während im Fall (b) dem Graphen eine neuer Shortcut hinzugefügt wird, mit den Kosten $cost((u, w)) = cost((u, v)) + cost((v, w))$.

2.3.2 Dijkstra mit Contraction Hierarchies

Wie bereits angesprochen, hat eine CH das Ziel die Suche nach einem kürzesten Pfad zu beschleunigen. Um die Vorteile einer CH nutzen zu können, muss der Dijkstra-Algorithmus (siehe Algorithmus 2.1) an die CH angepasst werden. Fortlaufend bezeichnen wir diese Variante des Dijkstra-Algorithmus als CH-Dijkstra. Der kürzeste Pfad von einem Startknoten s zu einem Zielknoten t wird nun bidirektional ermittelt. Man startet sowohl von s, als auch von t aus eine andere Variante des Dijkstra-Algorithmus. Hierbei werden vom Startknoten s aus nur ausgehende Kanten $(u, v) \in E$ relaxiert, bei denen $l(u) < l(v)$ ($l(u)$ = das Level des Knoten u). Auf der anderen Seite werden vom Zielknoten t aus nur eingehende Kanten (Rückwärtskanten) $(u, v) \in E$ relaxiert, bei denen $l(u) > l(v)$. Auf diese Weise berechnet man die kürzesten Pfade von s zu einem Knoten v und von v zu t. Während des CH-Dijkstras verwaltet man den gemeinsamen Knoten v mit der kleinsten Summe aus $d(s, v) + d(v, t)$, der bereits $bearbeitet$ wurde. Sobald ein Knoten $u \in PQ$ mit $d(s, u) > d(s, v) + d(v, t)$ oder $d(u, t) > d(s, v) + d(v, t)$ das vorderste Element in PQ ist, so bricht der Algorithmus ab, da es keinen anderen gemeinsamen Knoten mehr geben kann, über den der Pfad kürzer wäre als über v. Falls also ein kürzester Pfad von s nach t existiert, so verläuft er über einen gemeinsamen Knoten v, der von s und t besucht wurde und dessen Kosten $d(s, v) + d(v, t)$ minimal sind. Der kürzeste Pfad von s nach t ergibt sich schließlich aus dem kürzesten Pfad von s nach v und von v nach t. Damit nun aus dem gefundenen kürzesten Pfad ein Pfad generiert werden kann, der lediglich die Originalkanten des Ausgangsgraphen G enthält, müssen alle Shortcuts, die im Pfad enthalten sind, entpackt werden.

2.3.3 Hub-Labels

Eine weitere Technik zur Beschleunigung von kürzesten Pfadanfragen sind Hub-Labels [ADGW11]. Diese lassen sich aus einem gegebenen CH-Graph berechnen. Sie können benutzt werden, um die kürzeste Distanz $d(s,t)$ von einem Startknoten s zu einem Zielknoten t zu ermitteln. Die Berechnung der Hub-Labels basiert auf dem Level der einzelnen Knoten, daher hängt die Qualität der Hub-Labels stark von der Kontraktionsreihenfolge in der Vorverarbeitungsphase der CH ab.

Überblick

Bei einer kürzesten Pfadanfrage von s nach t auf einem Graph $G(V,E)$, werden durch den Dijkstra-Algorithmus in jedem Fall Teile des Graphen exploriert. Auf diese Weise lässt sich der kürzeste Pfad zwischen zwei Knoten ermitteln. Möchte man jedoch bei einer Suchanfrage lediglich die kürzeste Distanz $d(s,t)$ und nicht den kürzesten Pfad $p_{s,t}$) wissen, so kann dafür das Prinzip der Hub-Labels verwendet werden. Hierfür benötigt man einen Graphen, auf dem die Vorverarbeitungstechnik CH angewendet wurde (CH-Graph). Die Idee des "Hub-Labelling"besteht nun darin für jeden Knoten $v \in V$ ein $Label L(v)$ zu berechnen, sodass eine kürzeste Distanzanfrage für zwei gegebene Knoten $s,t \in V$ mittels dieser berechneten Labels $L(s)$ und $L(t)$ ermittelt werden kann. Mithilfe dieser Listen kann eine kürzeste Distanzanfrage um ein vielfaches schneller beantwortet werden, als mit einer herkömmlichen Dijkstra-Anfrage, da bei dieser Methode keine Kanten exploriert werden müssen und die Listen für jeden Knoten hinreichend klein sind.

Ein Label $L(v)$ sei definiert als eine Menge von 2-Tupel $(w,d(v,w))$ bzw. $(w,d(w,v))$ mit $v,w \in V$. Demnach verwaltet ein Label $L(v)$ eine Menge von Knoten w, die man auch als Hub bezeichnet, gepaart mit der kürzesten Distanz von v zu w oder umgekehrt von w zu v. Hierbei spielt die Art des Graphen eine wichtige Rolle. Je nach dem, ob es sich bei dem Graph um einen gerichteten oder ungerichteten Graph handelt, werden für jeden Knoten jeweils ein oder zwei Labels benötigt. Für ungerichtete Graphen reicht ein Label pro Knoten, da in einem ungerichteten Graphen $d(v,w) = d(w,v)$ gilt. Bei gerichteten Graphen ist dies nicht der Fall, daher werden hier zwei Labels pro Knoten benötigt. Da wir meistens mit gerichteten Graphen arbeiten, gehen wir ebenfalls von zwei Labels pro Knoten aus. Folglich berechnet man für jeden Knoten v ein Vorwärts-Label $L_f(v)$ (f = "forward") und ein Rückwärts-Label $L_r(v)$ (r = "reverse"). Das Label $L_f(v)$ besteht aus einer bestimmten Menge von 2-Tupel der Form $(w,d(v,w))$ und verwaltet somit eine bestimmte Menge von Knoten w zusammen mit der zugehörigen kürzesten Distanz von v nach w. Das Label $L_r(v)$ besteht aus einer bestimmten Menge von 2-Tupel der Form $(u,d(u,v))$ mit $u \in V$ und verwaltet damit eine bestimmte Menge von Knoten u zusammen mit der kürzesten Distanz von u nach v.

Ein $Hub - Label$ (HL) aus der Liste $L(v)$ sei definiert als ein 2-Tupel $(w,d(v,w))$ bzw. $(w,d(w,v))$ mit $w \in V$. Hierbei wird w als Hub und die zugehörige Distanz $d(v,w)$ bzw. $d(w,v)$ als $Hub - Distanz$ bezeichnet. Eine HL-Liste $L(v)$ verwaltet demnach für einen Knoten v eine bestimmte Menge an Knoten w und die zugehörige kürzeste Distanz

$d(v, w)$ bzw. $d(w, v)$. Je nach dem, ob es sich bei dem CH-Graph um einen gerichteten oder ungerichteten Graph handelt, werden für jeden Knoten jeweils eine oder zwei HL-Listen benötigt. Für ungerichtete Graphen reicht eine HL-Liste pro Knoten, da in einem ungerichteten Graphen $d(v, w) = d(w, v)$ gilt. Dies ist bei gerichteten Graphen nicht der Fall, daher benötigt man hier zwei HL-Listen pro Knoten benötigt. Da wir meistens mit gerichteten Graphen arbeiten, gehen wir von zwei HL-Listen pro Knoten aus. Dabei berechnet man für jeden Knoten v eine Vorwärts-HL-Liste $L_f(v)$ (f = "forward") und eine Rückwärts-HL-Liste $L_r(v)$ (r = "reverse"). Die HL-Liste $L_f(v)$ besteht aus einer Menge von HLs der Form $(w, d(v, w))$. Demnach verwaltet man in $L_f(v)$ eine Menge von Knoten w zusammen mit ihrer kürzesten Distanz $d(v, w)$ von v aus. Analog besteht die HL-Liste $L_r(v)$ aus einer Menge von HLs $(u, d(u, v))$. Demnach verwaltet man in $L_r(v)$ eine Menge von Knoten u zusammen mit ihrer kürzesten Distanz $d(u, v)$ zu v.

Kürzeste Distanzanfrage

Bei einer kürzesten Distanzanfrage von einem Knoten $s \in V$ zu einem Knoten $t \in V$ wird die Schnittmenge $L_f(s) \cap L_r(t)$ aller gemeinsamen Hubs v betrachtet. Die kürzeste Distanz von s nach t verläuft über einen dieser gemeinsamen $Hubs$. Folglich verläuft der kürzeste Pfad von s nach t über den gemeinsamen Hub v für den $d(s, v) + d(v, t)$ minimal ist. Daraus ergibt sich schließlich die kürzeste Distanz $d(s, v) + d(v, t)$. Sortiert man alle Labels $L(v)$ lexikographisch, so beträgt die Laufzeit einer kürzeste Distanzanfrage $O(|L_f(s)| + |L_r(t)|$.

Berechnung der Hub-Labels

Die Hub-Label-Algorithmus 2.4 mit dem die Labels für einen gegebenen CH-Graphen berechnet werden, fügt initial jedem Knoten $v \in V$ ein Paar $(v, 0)$ in beide Labels $L_f(v)$ und $L_r(v)$ hinzu. Nun iteriert man durch die Menge der Knoten V, wobei V in absteigender Reihenfolge nach Knotenlevel sortiert ist. Daher startet man bei dem Knoten $v \in V$, der das höchste Level in V besitzt. Sei v der zu betrachtende Knoten in jeder Iteration. Man iteriert nun durch alle Kanten von v und sucht dabei alle Knoten w mit $l(w) > l(v)$. Alle Einträge in $L(w)$ werden nun an v weitergegeben, wobei die jeweilige Distanz des 2-Tupels um die Kosten der Kante erhöht wird. Um es mit anderen Worten auszudrücken, holt sich ein "Kindknoten"(niedrigeres Level) von allen Ëlternknoten"(höheres Level) die Hubs und passt dabei die Distanzen entsprechend an. Auf diese Weise sind die Labels eines Knotens nach der Iteration vollständig gefüllt. Da es zwei Labels gibt, füllt man das entsprechende Label je nach Art der Kante. Handelt es sich um eine ausgehende Kante $(v, w) \in E$, so überträgt man alle Hubs aus $L_f(w)$ in $L_f(v)$ und erhöht dabei jeweils die Distanz um $cost((v, w))$. Analag überträgt man alle Hubs aus $L_r(w)$ in $L_r(v)$, wenn es sich um eine eingehende Kante $(w, v) \in E$ handelt. Auch hier wird die Distanz jeweils um die Kantenkosten $cost((w, v))$ erhöht.

Algorithmus 2.4 Hub-Label-Berechnung

```
function HubLabelling(V,E,cost)
    V.sortByLevel()                     // nach Level in absteigender Reihenfolge sortieren
    for all s ∈ V do
        s.Label ← add (s,0)
        for all Edge e = (s,t) ∈ E OR e = (t,s) ∈ E do
            if l(t) > l(s) then
                for all Hub h in t.Label do
                    if h.hub exists in s.Label then
                        update h.distance in s.Label to minimum from both
                    else
                        Hub newHub ← (h.hub, h.distance + cost(e))
                        s.Label.push(newHub)
                    end if
                end for
            end if
        end for
    end for
    s.Label.sortByHub()                 // nach hub in aufsteigender Reihenfolge sortieren
    for all s ∈ V do
        pruneHubs(V, s)
    end for
    return V, E, cost
end function
```

Algorithmus 2.5 Kürzen der Hubs

```
function pruneHubs(V, s)
    for all Hub h in s.Label do
        shortestDistance ← calculate shortest distance between s and h.hub
        if shortestDistance < h.distance then
            prune h from s.Label
        end if
    end for
end function
```

Da durch diese Vorgehensweise unnötige Hubs ebenfalls übertragen werden, müssen diese ausfindig gemacht und entfernt werden, sodass die Größe der Labels hinreichend klein bleibt und eine kürzeste Distanzanfrage schnell beantwortet werden kann. Beispielsweise kann es passieren, dass beim Übertragen der Hubs von einem Label $L(v)$ in ein anderes Label $L(u)$, das zu übertragende Hub bereits in $L(u)$ existiert, wodurch dieses Hub nach der Übertragung in $L(u)$ doppelt vorkommen würde. Um dies zu vermeiden, prüft man vor dem übertragen eines Hubs, ob das jeweilige Hub bereits existiert. Ist dies der Fall, so werden die jeweiligen Distanzen des 2-Tupels verglichen und der minimale von beiden übernommen. Nachdem alle Labels gefüllt wurden, kann man mit dem Kürzen der überflüssigen Hubs beginnen, dem sogenannten *prunen* (siehe Algorithmus 2.5). Beim *prunen* wird ein

Element $(w, d(v, w)$ aus der Liste $L_f(v)$ betrachtet. Dabei wird geprüft, ob die Distanz $d(v, w)$ die tatsächliche kürzeste Distanz zum Hub w ist. Um dies zu prüfen, startet man eine kürzeste Distanzanfrage von v zu dem zu prüfenden Hub w. Sollte ein anderer gemeinsamer Hub $u \in L_f(v) \cap L_r(w)$ existieren, bei dem $d(v, u) + d(u, w) < d(v, w)$ ist, so wird der Hub aus $L_f(v)$ entfernt (gekürzt), da $d(v, w)$ nicht die korrekte kürzeste Distanz von v zu w ist. Elemente $(w, d(w, v)$ im Label $L_r(v)$ lassen sich analog kürzen, indem eine kürzeste Distanzanfrage von w nach v gestartet wird und ebenfalls nach einem gemeinsamen Hub $u \in L_r(v) \cap L_f(w)$ gesucht wird, für den $d(w, u) + d(u, v) < d(w, v)$ gilt. Falls solch ein Hub u nicht existiert, so bleibt das Hub erhalten. Andernfalls wird es aus $L_r(v)$ entfernt.

Durch das Kürzen fallen etwa ein Viertel aller berechneten Hubs weg. Dadurch bleibt die Anzahl der Hubs pro Knoten relativ gering. Beispielsweise kann für einen Graphen mit ca. 120.000 Knoten eine Anzahl 10 Millionen Hubs berechnet werden. Dies entspricht etwa 85 Hubs pro Knoten. Implementiert man die kürzeste Distanzanfrage hinreichend effizient genug, so liegt die Antwortzeit einer Anfrage im Mikrosekundenbereich und ist damit um ein vielfaches schneller als eine Dijkstra-Anfrage.

3 Heuristiken zur Erstellung von Conctraction Hierarchies

3.1 Problemstellung

Gegeben sei ein gerichteter, positiv gewichteter Graph $G(V, E)$ mit der zugehörigen Kostenfunktion $cost : E \to \mathbb{R}^+$. Desweiteren sind zwei Tools gegeben. Beide Tools wurden im Rahmen einer Projektarbeit von Samuel Holderbach, Dominik Krenz und mir verknüpft und erweitert. Das Tool [] dient dabei zur Erstellung eines CH-Graph $G^{'}(V, E \cup E^{'})$ mit der zugehörigen Levelfunktion $l : V \to \mathbb{N}_0$ aus einem gegebenen Graph $G(V, E)$. Die Kontraktionsreihenfolge kann dabei selbst festgelegt werden. Dieses Tool bezeichnen wir fortlaufend als CH-Konstruktor. Mithilfe des anderen Tools [] können die Hubs eines CH-Graph berechnet werden. Außerdem kann der CH-Graph auf einem Globus visualisiert werden. Dabei ist es möglich CH-Dijkstra-Suchanfragen zu stellen und die entsprechenden CH-Dijkstra-Suchräume, sowie die berechneten Hubs des Start- und Zielknoten anzeigen zu lassen. Dieses Tool bezeichnen wir fortlaufend als den Visualisierer. Ziel der Arbeit ist es, mithilfe der gegebenen Tools verschiedene Strategien zur Bestimmung der Kontraktionsreihenfolge zu evaluieren. Dabei lässt sich die Qualität einer Strategie an der Anzahl der insgesamt berechneten Hubs bemessen, da die Berechnung und damit die Anzahl der Hubs stark von der Kontraktionsreihenfolge abhängen. Gesucht ist sind demnach eine Strategie zur Bestimmung einer Kontraktionsreihenfolge, bei der die Summe $\sum_{v \in V} |L_f(v)| + |L_r(v)|$ so gering wie möglich ist. Je geringer die Summe, desto höher ist die Qualität der Strategie. Sollte eine Strategie A diesbezüglich eine höhere Qualität aufweisen als eine andere Strategie B, so sollte ebenfalls die benötigte Berechnungszeit dieser Strategien verglichen werden. Falls die Strategie A eine längere Berechnungszeit aufweist, so sollte der dadurch entstandene ”Tradeofffëbenfalls evaluiert werden.

3.2 Hub-Score

3.2.1 Idee

Die Idee für dieser Heuristik entstand bei der Benutzung des Visualisierers. Bei vielen Start- Zielanfragen war es nämlich so, dass oft viele Hubs eines Knotens sich an einer Stelle gehäuft haben, wie man auch in Abbildung (3.1) erkennen kann. Diese Anhäufungen von Hubs in sehr engen Räumen war ein Indiz dafür, dass die gewählte Kontraktionsreihenfolge nicht ideal war. Ziel dieser Heuristik ist es, aus dem Haufen von Hubs einen Repräsentanten

zu wählen und ihm das höchste Level zu geben. Dadurch soll gewährleistet werden, dass aus ihm ein Hub wird, während die anderen keine Hubs mehr sind, wodurch die Anzahl der Hubs verringert wird. Abbildung (ABB) dient dabei zur Veranschaulichung. Jeder Hub w vergibt dabei einen $Score$ an Knoten u mit $l(w) > l(u)$ und $w, u \in V$. Diesen $Score$ bezeichnen wir fortan als Hub-Score.

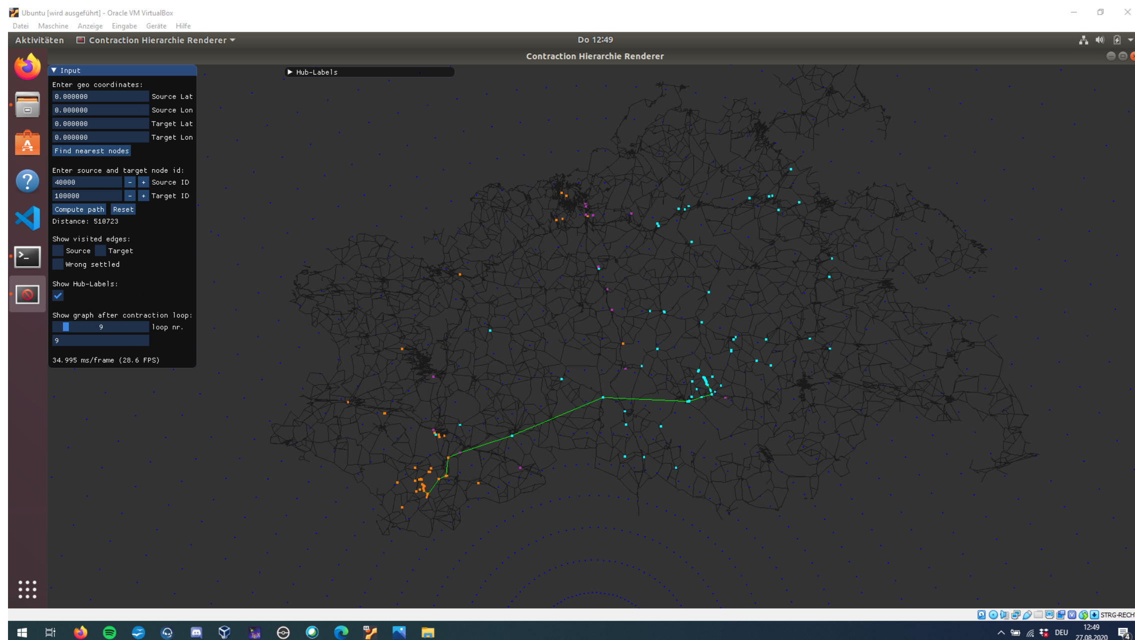


Abbildung 3.1: Screenshot aus Visualisier: Die blauen Rechtecke stehen für die hubs der Startknoten und die orangen Rechtecke stehen für die hubs der Zielknoten. Die lilalen Rehctecken entsprechen den gemeinsamen hubs.

3.2.2 Berechnung

Shortest-Path-Tree

Für die Berechnung der Hub-Scores, wird eine Variante des Dijkstra-Algorithmus verwendet. Man berechnet für jeden Knoten $s \in V$ seinen Shortest-Path-Tree $T(s)$. Der Shortest-Path-Tree ist ein Suchbaum der entsteht, wenn man einen Dijkstra-Algorithmus ohne einen Zielknoten startet. Diese spezielle Art des Dijkstra-Algorithmus nennen wir One-To-All-Dijkstra (OTA-Dijkstra). Er berechnet für einen gegebenen Startknoten s den kürzesten Pfad zu allen erreichbaren Knoten $t \in V$. Daher auch der Name One-To-All. Diese kürzesten Pfade sind alle in $T(s)$ enthalten, wodurch $T(s)$ einen Teilgraph von G darstellt. Dieser berechnete Shortest-Path-Tree wird nun für die Hub-Scoreverteilung verwendet. Kanten, die in einer Dijkstra-Anfrage besucht wurden, jedoch nicht zu einem kürzesten Pfad von s zu einem beliebigen Knoten $v \in V$ führen, nennt man Wrong-Settled-Edges. Diese Kanten sollen für die Scoreverteilung ignoriert werden. Es werden lediglich die Kanten von $T(s)$ verwendet.

Wie in Abschnitt Abschnitt 2.3.2 bereits erläutert, relaxiert der CH-Dijkstra lediglich Kanten (v, w), bei denen $l(v) < l(w)$ gilt. Demnach entsteht bei einem One-To-All-CH-Dijkstra ein Upwards-Shortest-Path-Tree, der sich vom normalen Shortest-Path-Tree unterscheidet, da nicht alle Kanten betrachtet werden. Da bei der Scoreverteilung ein Shortest-Path-Tree verwendet wird, ist das Ergebnis der Berechnung davon abhängig, welche Art von Shortest-Path-Tree verwendet wird. Für die Berechnung der Hub-Scores werden in dieser Arbeit beide Varianten getestet.

Scoreverteilung

Die eigentliche Idee ist es, zunächst für jeden Knoten $u \in V$ seinen Shortest-Path-Tree $T(u)$ zu berechnen. Basierend auf allen Hubs in $L(u)$ und auf $T(u)$ werden nun die Hub-Scores verteilt. Man iteriert durch das Label $L(u)$ und durchläuft für jeden Hub $w \in L(u)$ den kürzesten Pfad $p_{u,w}$ rückwärts von w nach u. Auf dem Weg verteilt man an jeden Knoten v, den man dabei besucht, einen Hub-Score. Hierbei gibt es zwei verschiedene Herangehensweisen, wie genau ein Score aussehen kann:

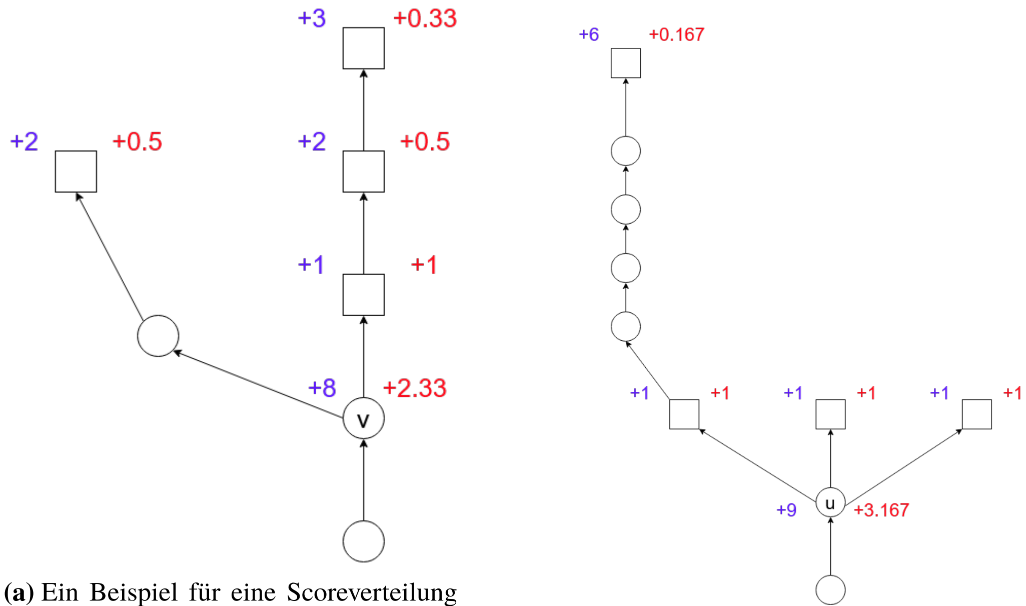
Ansatz 1

Wir nehmen an, dass je größer ein Score ist, desto wichtiger ist der Knoten. Demnach sollte man Knoten, die näher an einem Hub sind, einen größeren Score zuweisen als Knoten, die weiter vom Hub entfernt sind. Sei x die räumliche (Anzahl durchlaufener Kante) bzw. zeitliche (tatsächliche Kosten) Distanz zwischen einem Hub und einem besuchten Knoten. Dann würde ein Hub-Score mit der Formel $score = \frac{1}{x}$ berechnet werden. Je weiter entfernt ein Knoten ist, desto größer ist x, wodurch der Score $\frac{1}{x}$ kleiner wird.

Ansatz 2

Hier nehmen wir an, dass je kleiner ein Score ist, desto wichtiger ist der Knoten. Demnach sollte man Knoten, die näher an einem Hub sind, einen kleineren Score zuweisen als Knoten, die weiter vom Hub entfernt sind. Sei auch hier wieder x die räumliche / zeitliche Distanz zwischen einem Hub und einem besuchten Knoten. Dann würde ein Hub-Score mit der Formel $score = x$ berechnet werden. Je weiter entfernt ein Knoten ist, desto größer ist x, wodurch der Score x ebenfalls groß ist.

Anhand der Abbildung 3.2 lässt sich schnell feststellen, welcher Ansatz für die Scoreverteilung eher geeignet ist. In Ansatz 1 wird versucht, der eigentlichen Idee der Hub-Scores gerecht zu werden, da man Knoten mit einem guten Score verseht, die viele Hubs um sich herum haben. Dadurch wird aus diesen Anhäufungen von Hubs ein Repräsentant bestimmt, der später alle Hubs ersetzen wird. Falls außerhalb dieser Anhäufungen ebenfalls Hubs existieren, die dem Knoten ein Score zuweisen, so wird er zusätzlich geringfügig belohnt. Diese Belohnungen sind jedoch hinreichend klein, je größer die Distanz x ist. Das Problem an Ansatz 2 ist, dass Knoten bei der Scoreverteilung Knoten zu hart bestraft werden können. Dadurch wird die eigentliche Idee des Algorithmus nicht korrekt umgesetzt. Sobald man von einem weit entfernten Hub einen schlechten Score erhält, kann es sein, dass dadurch der falsche Repräsentant aus den Anhäufungen generiert wird, da sein Score mit steigendem x um ein vielfaches verschlechtert werden kann. In Ansatz 1 hingegen, wird ein Knoten in



(a) Ein Beispiel für eine Scoreverteilung an einen Knoten v. Die Rechteckte stellen Hubs dar. Die blauen Zahlen stehen für die Scoreverteilung nach Ansatz 2: Score = x, mit x als räumliche Distanz (Anzahl Kanten). Die roten Zahlen repräsentieren für Ansatz 1: Score = $\frac{1}{x}$.

(b) Ein weiteres Beispiel einer Scoreverteilung, bei der der Knoten u einen Score von 9 nach Ansatz 2 erhält, während er einen Score von 3.167 nach Ansatz 1 erhalten würde.

Abbildung 3.2: In den gezeigten Abbildungen hat der Knoten v nach Ansatz 1 einen Score von 2.33, während der Knoten u einen Score von 3.167 erhält. Knoten u hat demnach einen besseren Score als v. Nach Ansatz 2 hat v den besseren Score, obwohl Knoten u einen bessren Score haben müsste, da er der Idee des Algorithmus nach viele Hubs in direkter Nähe um sich hat.

solch einem Fall geringer belohnt. Dennoch wird er belohnt und nicht bestraft, was wiederum der Idee des Algorithmus nicht entgegenwirkt, da sein Score weiterhin geringfügig steigt und er damit relevant bleibt.

Geht man vom Upwards-Shortest-Path-Tree aus, so wird eine Art Normalisierung benötigt, da Knoten mit einem hohen Level automatisch einen schlechteren Score erhalten, da über diesen Knoten potentiell weniger Hubs existieren können. Dadurch wird relevanten Knoten mit einem hohen Level fälschlicherweise aus Prinzip ein schlechter Score gegeben. Um diesem Problem entgegenzuwirken wird eine weitere Variable zur Hub-Score Berechnung hinzugezogen: die Hubdichte. Sie ergibt sich aus der Anzahl Hubs in $L(v)$, sowie der Anzahl der Knoten im Shortest-Path-Tree $T(v)$: $hubDensity = \frac{|L(v)|}{|T(v)|}$.

3.3 Dijkstra-Score

3.3.1 Idee

Wie in Abschnitt 2.3 erwähnt, spiegelt das Level eines Knotens die Relevanz des Knotens wider. Je höher das Level, desto relevanter ist der Knoten. Aus dieser Tatsache heraus entstand die Idee für diesen Algorithmus. Ein Knoten über den viele kürzeste Pfade verlaufen wird als relevant angesehen. Offensichtlich ist in einem Straßennetzwerk eine Autobahn relevanter als eine Sackgasse, da über eine Autobahn mehr kürzeste Pfade verlaufen.

3.3.2 Berechnung

Auch in diesem Algorithmus wird ein OTA-Dijkstra verwendet. Man startet für jeden Knoten $v \in V$ ein OTA-Dijkstra. Nun werden alle kürzeste Pfade $p_{v,w}$ in $T(v)$ durchlaufen und dabei mitgezählt, wie oft ein Knoten u, in einem kürzesten Pfad $p_{v,w}$ vorkommt.

Betrachtet man einen kürzesten Pfad $p_{u,y} = u, v, w, x, y$, so sind $p_{u,x} = u, v, w, x$ und $p_{u,w} = u, v, w$ Teilpfade von $p_{u,y} = u, v, w, x, y$. Damit man diese Teilpfade nicht mehrfach durchläuft, kann man aus Gründen der Effizienz die Scores so verteilen, dass Teilpfade nicht mehrfach durchlaufen werden müssen. Dafür durchläuft man für jeden Blattknoten w $T(v)$ den kürzesten Pfad $p_{v,w}$ rückwärts von w nach v und inkrementiert den Score nach jeder durchlaufenen Kante. Auf diese Weise deckt man jeden Teilpfad von $p_{v,w}$ ab und beschleunigt somit den Prozess der Berechnung.

3.4 Heuristiken

Die ersten drei Heuristiken die gleich folgen, waren bereits im CH-Konstruktor gegeben, während die anderen im Rahmen dieser Arbeit hinzugefügt wurden. Alle Heuristiken gehen dem selben Schema nach. Es wird immer eine unabhängig Menge I (eine Teilmenge von Knoten $v \in G$, die nicht zueinander benachbart sind) von V berechnet. Die Berechnung von I ist dabei der wesentliche Unterschied zwischen den Heuristiken. Im Anschluss werden alle Knoten $v \in I$ kontrahiert. Alle Knoten, die in dieser Runde kontrahiert wurden, erhalten dasselbe Level. Das Level entspricht dabei der Kontraktionsrunde.

3.4.1 Quick-Contraction

Man beginnt mit einer sogenannten Quick-Contraction (QC). Hier werden in fünf Kontraktionsrunden jeweils die Knoten der berechneten Menge I kontrahiert. Dabei gilt für alle Knoten $v \in I$, dass ihre Anzahl an Kanten nicht größer als vier sein darf. Nach diesen fünf Kontraktionsrunden berechnet man weiterhin die Menge I, jedoch ohne die in der QC genannten Bedingung. Man kontrahiert so lange alle Knoten in I, bis jeder Knoten in G kontrahiert wurde.

3.4.2 One-By-One

Die One-By-One (OBO) Heuristik ist sehr schlicht gehalten. Wie der Name bereits verrät, wird nur ein Knoten pro Runde kontrahiert. Jede Runde wird der hinterste Knoten in V ausgewählt und kontrahiert.

3.4.3 Edge-Difference

Der Ablauf einer Kontraktionsrunde beginnt damit, dass die Knoten nach dem In-Out-Produkt (IO-Produkt) sortiert werden. Das IO-Produkt für ein Knoten v ergibt sich aus der Anzahl aller Kanten $(v, w) \in E$ multipliziert mit der Anzahl aller Kanten $(w, v) \in E$. Erst nachdem V nach diesem Kriterium in aufsteigender Reihenfolge sortiert wurde, wird I berechnet. Im Anschluss berechnet man die Edge-Difference (ED) aller Knoten in I. Nach der Kontraktion eines Knotens fallen seine Kanten weg, während möglicherweise neue Shortcuts dabei entstehen. Um die ED zu berechnen simuliert man die Kontraktion eines Knotens, um die Differenz der Kantenzahl in E, nach der Kontraktion festzustellen.

Edge-Difference = Anzahl neu einzufügender Shortcuts - Anzahl wegfallender Kanten

Nachdem die ED für jeden Knoten berechnet wurde, filtert man aus I alle Knoten, dessen ED größer als der Durchschnittswert aller EDs in I ist. Diese übrigen Knoten in I werden in dieser Runde kontrahiert. Man kontrahiert dabei immer den Knoten mit der minimal ED zuerst.

3.4.4 Score

Diese Heuristik verwendet die berechneten Scores aus Abschnitt 3.2 bzw. 3.3. Zunächst sortiert man die Menge aller V aufsteigend nach ihren Scores. Im nächsten Schritt berechnet man die Menge I. Im Anschluss berechnet man die EDs aller Knoten in I und (wie in der obigen Heurikstik) filtert danach alle Knoten aus I, dessen EDs größer als der Durchschnittswert aller EDs in I ist. Diese Knoten werden in dieser Runde kontrahiert. Dabei kontrahiert man immer den Knoten mit der minimalen ED zuerst.

3.4.5 Score-And-Edge-Difference

Die Vorgehensweise dieser Heuristik ist eine Kombination aus der Score- und der Edge-Difference-Heuristik. Man beginnt (wie bei der Edge-Difference-Heuristik) damit, die Knoten nach ihrem IO-Produkt in aufsteigender Reihenfolge zu sortieren. Danach berechnet man I und filtert I wieder nach dem Durschnittswert aller EDs. Die übrigen Knoten in I werden nun sowohl noch Score als auch nach ED sortiert. Um die Knoten nach beiden Kriterien fair sortieren zu können, muss man beide Werte auf den selben Bereich abbilden. Sei $sc(v)$ definiert als der Score von v und $ed(v)$ als die ED von v. Man berechnet nun für die übrigen Knoten $v \in I$ den Gesamtwert $sc(I)$ aller $sc(v)$ und den Gesamtwert $ed(I)$ aller $ed(v)$. Teilt man jetzt den Wert $sc(v)$ eines einzelnen Knotens mit dem Gesamtwert

$sc(I)$, so erhält man eine Zahl zwischen 0 und 1. Analog für die EDs. Im letzten Schritt addiert man beide Werte $\frac{sc(v)}{sc(I)} + \frac{ed(v)}{ed(I)}$. Der Knoten mit dem minimalen Wert wird zuerst kontrahiert.

Als Zusatz kann man jedem Kriterium noch eine Gewichtung geben. Beispielsweise kann es sinnvoll sein, den EDs mehr Gewichtung als den Scores zu geben. Dafür führt man eine weitere Variable r ein, die das Verhältnis bestimmt:

$r \cdot \frac{sc(v)}{sc(I)} + (1 - r) \cdot \frac{ed(v)}{ed(I)}$.

4 Messungen und Evalutaion

Die in diesem Abschnitt gezeigten Ergebnisse wurden auf einer Ubuntu 18.04 VM berechnet. Der verfügbare RAM betrug 5GB und die Anzahl der CPU Kerne 4, mit einer Taktfrequenz von 1.6GHz pro Kern.

4.1 Testgraphen und Durchführung

In Tabelle 4.6 werden alle Daten der verwendeten Graphen aufgeführt. Alle durchgeführten Tests fanden auf diese Graphen statt.

	LM	**BR**	**MV**
$\|V\|$	2828	119989	644199
$\|E\|$	4020	227567	1305996

Tabelle 4.1: Testgraphen - BR = bremen, LM = lower Manhattan, MV = Mecklenburg-Vorpommern

Zunächst wurden alle Heuristiken, so weit es möglich war, auf den Graphen getestet. Die OBO Heuristik rechnete selbst auf dem bremen-Graph ewig und wurde deshalb verworfen. Dies kann auch an der Hardware liegen, da die gegebene Umgebung nicht ganz optimal ist. Auch für den MV-Graphen war die Berechnungszeit nicht mehr tragbar, weswegen nicht alle Heuristiken auf ihm getestet wurden. Dennoch kann man am LM-Graph gut erkennen, dass die OBO Heuristik im Vergleich zu anderen die dreifache Menge an Hubs zur Folge hat. Aufgrund des schlechten Ergebnisses, sowie der langen Berechnungszeit, kann diese Heuristik verworfen werden. Überraschenderweise liefert die QC ein sehr gutes Ergebnis ab. Die geringe Anzahl an Hubs, sowie die geringe Berechnungszeit bei der Erstellung der CH, machen diese Heuristik zu einer der Besten der fünf vorgestellten. Das beste Ergebnis für die LM-Graphen lieferte die ED-Heuristik, während man das beste Ergebnis für den BR-Graphen mithilfe der Score-Heuristik erreicht hat. Für die Score Heuristik nutzte man die berechneten Dijkstra-Scores. Ein weiterer überraschender Aspekt sind die Anzahl der Kanten. Diese blieben relativ konstant und veränderten sich kaum, ungeachtet dessen, welche Heuristik angewendet wurde. Die Berechnungszeit ist ein eher irrelevanter Faktor, da sowohl die Berechnung der Hubs, als auch die Erstellung eines CH, meistens im Sekundenbereich lag.

	LM	BR	MV
Anz. bwd Hubs	152165	-	-
Anz. fwd Hubs	123115	-	-
Anz. Gesamt	275280	-	-
Anz. Kanten	11901	-	-
Berechnungszeit der Hubs	0.0021721 min	-	-

Tabelle 4.2: OBO

	LM	BR	MV
Anz. bwd Hubs	49340	4849410	35392293
Anz. fwd Hubs	48033	5015961	35536630
Anz. Gesamt	97373	9865371	70928923
Anz. Kanten	7772	400801	2314020
Berechnungszeit der Hubs	0.00073798 min	0.0945807 min	0.880974 min

Tabelle 4.3: QC

	LM	BR	MV
Anz. bwd Hubs	46587	5025027	37433057
Anz. fwd Hubs	48102	5037684	37226332
Anz. Gesamt	94689	10062711	74659389
Anz. Kanten	7625	3996641	2313459
Berechnungszeit der Hubs	0.000724942 min	0.128465 min	0.973581 min

Tabelle 4.4: ED

	LM	BR	MV
Anz. bwd Hubs	50366	4920560	-
Anz. fwd Hubs	47076	4596303	-
Anz. Gesamt	97442	9516863	-
Anz. Kanten	7852	403560	-
Berechnungszeit der Hubs	0.000775296 min	0.139744 min	-

Tabelle 4.5: SC

In den Tabellen 4.7 und 4.8 ist das Ergebnis eines Versuchs, einen CH-Graphen mithilfe der Scores zu verbessern. Verwendet wurde dafür der LM-CH-Graph, der durch die ED-Heuristik entstand. Zunächst wurden vier Score-Dateien berechnet. Für zwei Score-Dateien wurde der OTA-Dijkstra verwendet, während für die anderen beiden der OTA-CH-Dijkstra

	LM	BR	MV
Anz. bwd Hubs	46587	5025027	-
Anz. fwd Hubs	48102	5037684	-
Anz. Gesamt	94689	10062711	-
Anz. Kanten	7625	399641	-
Berechnungszeit der Hubs	0.000708264 min	0.0989036 min	-

Tabelle 4.6: SCED

verwendet wurde. Im Anschluss hat man jeweils die Scores in Abhängigkeit von der räumlichen, sowie der zeitlichen Distanz verteilt. Die zeitliche Distanz auf dem gesamten shortest-path-tree erwies für die Score-Heuristik als erfolgreich. Die Anzahl der Hubs wurde von 94689 auf 79291 was eine Verminderung von fast 17% darstellt.

	HS-E	HS-D	HS-E-Up	HS-D-Up
Anz. bwd Hubs	46739	39674	45342	44454
Anz. fwd Hubs	45673	39617	39631	43010
Anz. Gesamt	92412	79291	84973	87464
Anz. |E|	7755	7966	7807	7873

Tabelle 4.7: LM-CH-ED-SC: Score-Heurstik wurde auf den LM-CH-Graph, der mithilfe der ED bestimmt wurde, angewendet. HS-E = Hub-Score nach Anzahl Kante auf OTA-Dijkstra; HS-E-Up = analag für OTA-CH-Dijkstra

	HS-E	HS-D	HS-E-Up	HS-D-Up
Anz. bwd Hubs	46587	46587	46587	46587
Anz. fwd Hubs	48102	48102	48102	48102
Anz. Gesamt	94689	94689	94689	94689
Anz. |E|	7625	7625	7625	7625

Tabelle 4.8: LM-CH-ED-SCED: Score-And-Edge-Difference-Heurstik wurde auf den LM-CH-Graph, der mithilfe der ED bestimmt wurde, angewendet.

5 Zusammenfassung und Ausblick

Im Laufe dieser Bachelorarbeit wurden verschiedene Heuristiken zur Konstruktion einer Contraction Hierarchie vorgestellt. Des Weiteren wurde ein Algorithmus zur Verbesserung einer bestehenden CH implementiert und auf verschiedenen Ausgangsgraphen getestet und evaluiert. Zusammenfassend kann sagen, dass jede Heuristik verschiedene Ergebnisse geliefert hat. Leider konnte aufgrund der gegebenen Rechenleistungen nicht all zu große Graphen hinzugezogen werden.

Ausblick

Contraction Hierachies stellen eine geeignet Technik zur Verbesserung von Graphen dar. Das Gebiet gilt weiterhin größtenteils als unerforscht und bietet daher reichlich Möglichkeiten zur Erweiterung und Verbesserung an. Das Finden einer optimalen Kontraktionsreihenfolge ist in der Theorie sogar NP-schwer, weswegen man in der Praxis verschiedene Ansätze für Heuristiken entwickelt und diese entsprechend auswertet. Da die Implementierung des CH-Konstruktors hinreichend generisch umgesetzt wurde, lassen sich die Heuristiken ganz leicht hinzufügen und das Ergebnis auswerten. Bestehende Algorithmen, wie die Scoreverteilungsalgorithmen, können erweitert und effizienter umgesetzt werden, sodass sie auf größeren Graphen eine halbwegs effiziente Anwendung finden. Beispielsweise sind die Verminderung der Hubs von 17% ein Erfolg. Diese Heuristik könnte man auf sehr großen Graphen testen und das Verhalten darauf analysieren.

Literaturverzeichnis

[] *CH-Konstruktor*. URL: https://theogit.fmi.uni-stuttgart.de/nusserae/chconstructor (zitiert auf S. 25).

[] *Simple Graph Renderer*. URL: https://github.com/invor/simplestGraphRendering (zitiert auf S. 25).

[ADGW11] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. „A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks“. In: *International Symposium on Experimental Algorithms* 27 (2011), S. 1 (zitiert auf S. 20).

[Del08] R. G. P. S. D. S. D. Delling. *Contraction hierarchies: Faster and simpler hierarchical routing in road networks*. Experimental Algorithms, 2008 (zitiert auf S. 16).

[Dij59] E. W. Dijkstra. *A note on two problems in connexion with graphs*. Numerische Mathematik 1.1, 1959 (zitiert auf S. 15).

Alle URLs wurden zuletzt am 27. 08. 2020 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift