Karlsruhe Institute of Technology

Effiziente Berechnung von Alternativrouten mit der Penaltymethode und CH-Potentialen

Bachelorarbeit
von

Max Willich

An der Fakultät für Informatik
Institut für theoretische Informatik

Erstgutachter:	PD Dr. Torsten Ueckerdt
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Tim Zeitz, M.Sc.

Bearbeitungszeit: 02.11.2020 – 26.03.2021

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 26. März 2021

Abstract

This thesis revolves around the calculation of alternative routes between two points on road networks. Using the penalty method, a graph can be reduced to a so called *alternative graph*. A good alternative graph only has few edges, but many good alternative routes between the starting point and end point. In 2013, Kobitzsch et al. published their implementation of the penalty method, which calculates very good alternative graphs in only a few hundred milliseconds. We evaluate an alternative version of the penalty method using the A*-Algorithm and CH-Potentials, which is easier to implement, regarding graph quality and runtime. We achieve high success rates and high path qualities for all test instances. While our method achieves worse runtimes than that of Kobitsch et al., runtime generally stays below one second for all but the most difficult queries.

Deutsche Zusammenfassung

Wir beschäftigen uns mit der Berechnung von Alternativrouten von A nach B auf Straßengraphen. Mithilfe der Penalty-Methode können Graphen auf sogenannte Alternativgraphen reduziert werden. Ein guter Alternativgraph ist sehr dünn besetzt, beinhält aber viele gute Alternativrouten. Kobitzsch et al. veröffentlichten 2013 in [KRS13] eine Implementation der Penalty-Methode, welche sehr gute Alternativgraphen innerhalb weniger hundert Millisekunden berechnen kann. Wir evaluieren eine alternative Realisierung mithilfe des A*-Algorithmus und CH-Potentialen, welche einfacher zu implementieren ist, auf Qualität und Laufzeit. Wir erreichen hohe Erfolgsquoten und Pfadqualitäten für alle Testinstanzen. Wir erzielen schlechtere Laufzeiten als Kobitzsch et al., aber dennoch Laufzeiten unter einer Sekunde für alle außer die längsten Anfragen.

Inhaltsverzeichnis

1. Einleitung

Vorbei sind die Zeiten an denen man während der Autofahrt auf eine Straßenkarte gucken muss um den Weg zum Ziel zu finden. Heutzutage besitzt fast jeder Mensch ein Smartphone mit eingebauter Routenplanungssoftware, wie z.B. Google Maps oder Apple Maps. Nutzer haben dabei ein Ziel: Eine möglichst gute Route von A nach B zu finden. Aber wie funktioniert das?
Routenplanung ist ein Forschungsgebiet der Informatik, das sich mit der Suche nach Routen von A nach B auf realen Straßennetzen beschäftigt. Meistens wird dabei die schnellste Route gesucht. Grundlage der Routenplanung ist die Modellierung von Straßennetzen als Graphen: Kreuzungen werden zu Knoten eines Graphen und Straßen zwischen den Kreuzungen zu Kanten. Aufgrund der Vielzahl an Straßen und Kreuzungen in der gesamten Welt sind diese Straßengraphen entsprechend groß. Allein der Graph zum westeuropäischen Straßennetz besitzt über 10 Millionen Knoten und über 20 Millionen Kanten. Auf Graphen dieser Größenordnung ist die Berechnung der schnellsten Route zwischen zwei Knoten in Echtzeit keine triviale Aufgabe. Dijkstra's Algorithmus, ein schneller Algorithmus zur Berechnung von kürzesten Routen auf beliebigen gewichteten Graphen, benötigt für eine solche Anfrage auf dem Straßennetz von Europa mehrere Sekunden. Dies kann restriktiv sein, falls mehrere Kürzeste-Wege-Anfragen benötigt werden oder z.B. ein Routenplanungsserver mehrere Duzend Anfragen pro Sekunde erhält.
Dieses Problem wird gelöst, indem man die spezielle Struktur von Straßennetzen ausnutzt. In jedem Straßennetz gibt es eine klare Hierarchie der Straßen: Autobahnen sind schneller als Landstraßen und Landstraßen sind schneller als Stadthauptstraßen. Wenn man also von Hamburg nach Berlin fahren möchte, geht die schnellste Route sehr wahrscheinlich über eine Autobahn, die die beiden Städte verbindet. Mithilfe der Ausnutzung dieser Hierarchien ist es möglich, die Kürzeste-Wege-Anfragen auf Straßengraphen in unter einer Millisekunde zu beantworten.
Nachdem also das Problem, die schnellste Route zu finden, gelöst wurde, suchen wir nun nach Alternativrouten. Nicht immer ist die schnellste Route auch die vom Nutzer bevorzugte Route. Viele Autofahrer wollen z.B. nicht durch große Städte fahren, da man dort achtsamer sein muss als wenn man übers Land fährt. Einige Autofahrer vermeiden es, auf Autobahnen zu fahren. Und manche Autofahrer fahren lieber kurvige Strecken für erhöhten Fahrspaß.
Wie man sieht, ist es nicht möglich jede Vorliebe von Anwendern zu berücksichtigen. Anstatt also kürzeste Routen zu suchen, die Autobahnen oder große Städte vermeiden, suchen wir allgemeine Alternativrouten ohne externe Bedingungen. Der Nutzer kann sich dann die Route aussuchen, die für ihn am angenehmsten ist. Dabei stellt sich die Frage, was

eigentlich eine gute Alternativroute ist. Betrachten wir nochmal das Beispiel der Route von Hamburg nach Berlin: Die zweitschnellste Route ist vermutlich sehr ähnlich zur schnellsten Route, allerdings fährt man einmal kurz von der Autobahn auf eine Raststätte nur um dann direkt wieder auf die Autobahn zu fahren. Die drittschnellste ist dann wahrscheinlich wie die zweitschnellste Route, allerdings wird eine andere Raststätte angesteuert. Es gibt aber dennoch sinnvolle Alternativen: Während die kürzeste Route meistens eindeutig über die A24 ist, da diese direkt von Hamburg nach Berlin fährt, ist es auch möglich erst über die A7 zu fahren und dann in Hannover auf die A2 zu wechseln, welche nach Berlin führt. Aufgrund von beispielsweise Bauarbeiten an der A24 bevorzugen einige Autofahrer sicherlich die zweite Variante.
Es zeigt sich, dass die x-schnellsten Routen nicht unbedingt gute Alternativrouten sind. Um das Problem systematisch angehen zu können, werden einige Anforderungen an Alternativrouten gestellt. Eine Alternativroute muss eine gewisse Mindestabweichung von der schnellsten Route haben, eine Alternativroute darf nicht viel länger sein als die schnellste Route und jedes Teilstück einer Alternativroute darf auch nicht viel länger sein als die schnellste Route zwischen Anfang und Ende des Teils. Durch solche Bedingungen werden Alternativrouten ausgeschlossen, die offensichtlich nicht sinnvoll sind.

1.1 Verwandte Werke

Es ist nicht sinnvoll, solange die k-schnellste Route zu berechnen, bis man eine Route findet, die den Bedingungen entspricht. Bis man eine nennenswerte Abweichung von der schnellsten Route erreicht hat man schon mehrere hunderttausend Routen berechnet. Dies resultiert in einer entsprechend langsamen Laufzeit. Weiterhin suchen wir nicht nur eine Alternativroute, sondern gleich eine ganze Menge von Alternativrouten, damit der Nutzer die maximale Auswahl hat.
Via-Node-Routing ([ADGW13]):
Ein besserer Ansatz ist die *Via-Node*-Methode. In diesem werden kürzeste Routen gesucht, die über sogenannte *Via-Knoten* v gehen, also die kürzeste Route vom Startknoten bis v und dann die kürzeste Route von v zum Zielknoten. Im Beispiel von Hamburg nach Berlin wäre ein sinnvoller Via-Knoten z.B. ein Knoten auf der A7, denn die kürzeste Route von Hamburg nach Berlin, die über die A7 gehen muss, wäre die vorhin beschriebene Route über die A2.
Es stellt sich nun die Frage, wie man sinnvolle Via-Knoten findet. Wenn man jeden Knoten im Straßengraphen betrachten würde, müsste man wieder Millionen von Kürzeste-Wege-Anfragen stellen. Dies ist viel zu langsam. Bader et al. stellen in [BDGS11] drei Algorithmen vor, die sinnvolle Via-Knoten berechnen. Diese Algorithmen heißen X-BDV, X-REV und X-CHV. Alle drei Algorithmen suchen nach Alternativrouten mit vier Anforderungen:

- Die Alternativroute ist maximal $\epsilon = 10\%$ länger als die schnellste Route
- Die Alternativroute teilt maximal $\gamma = 80\%$ ihrer Strecke mit der schnellsten Route
- Jede Teilstrecke der Alternativroute, die kürzer ist als $\alpha = 25\%$ der Gesamtlänge, ist eine schnellste Route
- Jede Teilstrecke, die länger ist als $\alpha = 25\%$ der Gesamtlänge, ist maximal $\epsilon = 10\%$ länger als die schnellste Route zwischen den Teilenden

Die zweite Bedingung sorgt für nennenswerte Abweichungen von der schnellsten Route. Die dritte und vierte Bedingung sorgt dafür, dass Alternativrouten keine sinnlosen Umwege fahren.
X-BDV ist zwar für reelle Anwendungen zu langsam und benötigt bis zu einer Minute für eine Anfrage, aber findet eine große Anzahl von Pfaden mit sehr guter Qualität. Er

wird also vor allem als Richtwert für die Pfadqualität genutzt, an der man einen anderen Algorithmus messen kann. Weiterhin kann man X-BDV für kleinere Graphen nutzen. X-REV und X-CHV nutzen dagegen die hierarchischen Strukturen von Straßennetzen aus um klare Laufzeitverbesserungen zu erzielen. Beide Algorithmen benötigen nur wenige Millisekunden für eine Anfrage und finden trotzdem häufig mindestens eine Alternativroute, die die geforderten Bedingungen erfüllt.

Penalty-Methode ([BDGS11], [KRS13]):
Ein anderer Ansatz, alternative Routen zu berechnen ist, den kürzesten Weg zu suchen und zu verschlechtern, indem man die Kantengewichte des kürzesten Pfades erhöht. Der neue kürzeste Pfad ist dann, je nachdem wie stark die Kantengewichte erhöht wurden, mehr oder weniger disjunkt vom alten kürzesten Pfad. Wiederholt man dieses Query-Penalty-Schema mehrfach, erhält man zunehmend längere, aber auch disjunktere Pfade.
Die Penalty-Methode nutzt den Ansatz von oben. Allerdings werden die gefundenen Pfade in jeder Iteration nicht direkt als Alternativpfad genutzt, sondern in einen *Alternativgraphen* $\mathcal{A}$ hinzugefügt. Nach einer gewissen Menge von Iterationen bricht die Penalty-Methode dann ab und es können andere, für Straßengraphen eigentlich zu langsame, Algorithmen wie X-BDV verwendet werden, um Alternativrouten aus dem Graphen zu extrahieren. Das hat den Vorteil, dass die X-BDV-Qualitätskriterien eingehalten werden können. Weitere Details finden sich im Grundlagenkapitel.

2. Grundlagen

2.1 Definitionen

Ein Graph $G = (V, E)$ ist ein 2-Tupel aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$. Ein Knoten $v \in V$ heißt zu einem anderen Knoten w *adjazent*, wenn $(v, w) \in E$. Zur besseren Lesbarkeit schreiben wir für eine Kante (u, v) auch uv. Ein Graph heißt *ungerichtet*, wenn es für jede Kante uv auch eine Kante vu gibt. Die Kardinalität der Knotenmenge V nennen wir N, die Kardinalität der Kantenmenge E nennen wir M.
Eine Metrik w ist eine Funktion $w : E \to \mathbb{N}_0$. Den Wert $w(e)$ nennen wir *Gewicht* oder *Kosten* der Kante e. Für diese Arbeit lassen wir nur nichtnegative Kantengewichte zu.
Ein *Pfad* ist eine Folge von Knoten $(v_0, v_1, \ldots, v_n)$. Wir nennen die Menge aller Pfade eines Graphens G $\mathcal{P}_G$. Die Kantenmenge E_P eines Pfades P ist die Menge aller Kanten, dessen Endpunkte in P liegen. Wir schreiben statt $e \in E_P$ zur Vereinfachung $e \in P$. Wir nennen $len_w(P) = \sum w(e) : e \in P$ die *Länge* eines Pfades P bezüglich der Metrik w. Wir bezeichen mit P_u^v, wobei $u, v \in P$, den Teilpfad von u nach v über die Kanten von P.
Für zwei Knoten s und t eines Graphen G existiert mindestens ein Pfad mit minimaler Länge. Diesen Pfad nennen wir *kürzesten Pfad*. Wir definieren $\pi : V \times V \to \mathcal{P}$ als die Funktion, die zwei Knoten s und t auf den kürzesten Pfad von s nach t abbildet. $\mathcal{D}_w : V \times V \to \mathbb{N}_0$ sei die Funktion, die zwei Knoten s und t auf die Länge von $\pi(s, t)$ abbildet. Für unsere Arbeit setzen wir für unsere Graphen vorraus, dass sie ungerichtet und stark zusammenhängend sind, also dass es einen Pfad von jedem Knoten zu jeden anderen Knoten gibt. Weiterhin haben unsere Graphen keine Doppelkanten.
Wir definieren für einen Alternativpfad P von s nach t die folgenden drei Qualitätsmerkmale:

1. Sharing: $\sum_e w(e) : e \in \{e \mid e \in P \wedge e \in \pi(s, t)\}$
2. Uniformly Bounded Stretch: $\max\{\frac{len_w(P_u^v)}{\mathcal{D}_w(u,v)} : u, v \in P\}$
3. Local Optimality: $\max\{\frac{len_w(P_u^v)}{\mathcal{D}_w(s,t)} : u, v \in P,\ len_w(P_u^v) \neq \mathcal{D}_w(u, v)\}$

Sharing gibt an, welchen Anteil sich der Alternativpfad mit dem kürzesten Pfad teilt. Uniformly Bounded Stretch gibt die Obergrenze für die Länge eines Teilpfades vom Alternativpfad im Verhältnis zur kürzesten Distanz zwischen den Teilpfadenden an. Local Optimality gibt an, bis zu welcher Länge alle Teilpfade die kürzesten Pfade zwischen ihren Enden sind. Uniformly Bounded Stretch kürzen wir mit *UBS* ab, Local Optimality mit *LO*.

2.2 Dijkstra's Algorithmus [Dij59]

Eingabe: Ein Graph $G = (V, E)$ und ein Startknoten s
Ausgabe: Kürzeste Wege $\{P_{v_0}, \ldots, P_{v_N}\}$ von s zu jedem Knoten $v_i \in V$ auf G.

Dijkstra's Algorithmus löst das Kürzeste-Wege-Problem in polynomieller Laufzeit. Für jeden Knoten speichert Dijkstra's Algorithmus die bislang gefundene Distanz und dessen *Vorgängerknoten*. Diese Distanz nennen wir *tentative Distanz*. Wir bezeichnen die Funktion, die einen Knoten auf dessen tentative Distanz abbildet mit $d : V \to \mathbb{N}_0 \cup \infty$, und die Vorgängerfunktion als $p : V \to (V \cup \bot)$
Weiterhin nutzt Dijkstra's Algorithmus eine *Prioritätswarteschlange*. In einer Prioritätswarteschlange liegen Elemente der Form $(v, d) \in V \times (\mathbb{N}_0 \cup \infty)$. d nennen wir hier den *Schlüssel* des Knotens v. Eine Prioritätswarteschlange stellt mindestens folgende Funktionen bereit:

- `pop`: Gibt das Element mit dem niedrigsten Schlüssel aus und löscht es aus der Schlange.
- `push (v, d)`: Fügt das Tupel (v, d) in die Schlange ein.
- `decrease-key (v, d)`: Verringert den Schlüssel von v auf d.

Das Tupel $(s, 0)$ wird in die Prioritätswarteschlange hinzugefügt. Die Distanzen aller Knoten außer s wird auf ∞ gesetzt, die Distanz von s auf 0. Der Vorgänger aller Knoten wird auf $\bot$ gesetzt. Jetzt wird folgendes Verfahren wiederholt, bis die Schlange leer ist:

1. Erhalte Tupel (v, d) durch `pop`
2. Betrachte alle ausgehenden Kanten $\{e_0, \ldots, e_n\}$ von v. Sei v_i der Zielknoten von Kante e_i.
3. Falls $d(v) + c(e_i) < d(v_i)$:
   - Setze $d(v_i) = d(v) + c(e_i)$ und $p(v_i) = v$.
   - Falls v_i nicht in der Schlange enthalten ist, füge $(v_i, d(v_i))$ in die Schlange hinzu
   - Ansonsten: Verringere den Schlüssel von v_i auf $d(v_i)$.

Die Distanz von s zu v ist nach Abschluss des Algorithmus durch $d(v)$ ablesbar. Um den kürzesten Pfad von s nach v zu finden, geht man von v aus immer die Vorgängerknoten entlang, bis man nach v gekommen ist.
Die Laufzeit von Dijkstra's Algorithmus ist von der Laufzeit von `decrease-key` und `push` abhängig. Wenn man für die Prioritätswarteschlange einen binären Heap verwendet, ist die Worst-Case-Laufzeit von Dijkstra's Algorithmus in $\mathcal{O}((N + M) \log(N))$.
Wenn man nur den kürzesten Pfad von s zu einem Knoten t sucht, kann der Algorithmus abgebrochen werden, sobald t von der Schlange entfernt wird. Ab da kann sichergestellt werden, dass der kürzeste Pfad für t gefunden wurde.

2.3 A*-Algorithmus [HNR68]

Eingabe: Ein Graph $G = (V, E)$ mit Metrik w, Startknoten s, Zielknoten t und Heuristik $h : V \to \mathbb{N}_0$
Ausgabe: Kürzester Pfad von s nach t auf G.

Der A*-Algorithmus ist eine Erweiterung von Dijkstra's Algorithmus. Zusätzlich zum Graphen G und dem Startknoten s bekommt der A*-Algorithmus noch einen Zielknoten t und eine Heuristik $h : V \to \mathbb{N}_0$, welche abschätzt, wie weit ein Knoten von t entfernt ist. Für diese Arbeit nehmen wir an, dass folgende Bedingungen für h gelten müssen:

- $h(v) \leq \mathcal{D}_w(v,t)$
- Für $e = (v,w) \in E$: $h(w) \leq h(v) + c(e)$

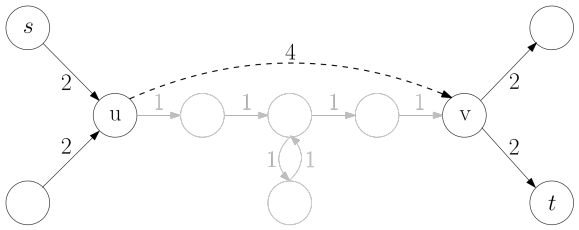
Eine Heuristik h, die diese Bedingungen erfüllt, nennen wir *zulässig*.
Es folgt, dass der A*-Algorithmus nur den kürzesten Weg von s zu einem Knoten t, nicht aber zu jeden anderem Knoten in V berechnen kann. Dieses Problem nennt man auch Punkt-zu-Punkt-kürzester-Weg-Problem.
Wie Dijkstra's Algorithmus unterhält auch der A*-Algorithmus eine Prioritätswarteschlange sowie eine Distanz- und eine Vorgängerfunktion (wie oben beschrieben). Weiterhin wird eine Liste aller Knoten geführt, die von dem Algorithmus fertig abgearbeitet worden sind. Diese Liste nennen wir *Closed List*. In der Literatur wird die Warteschlange auch oft als *Open List* bezeichnet. Die Menge aller Knoten in der *Closed List* nennen wir *Suchraum* von A*.
Um A* zu initialisieren, füge $(s, h(s))$ in die Schlange ein. Setze $d(v) = \infty$ für alle Knoten außer s und $d(s) = h(s)$. Setzte $p(v) = \perp$ für jeden Knoten $v \in V$. Führe dann folgendes Verfahren aus, bis dieses abbricht oder die Schlange leer ist:

1. Erhalte Tupel (v,d) durch `pop`
2. Falls v der Zielknoten t ist, breche ab.
3. Füge v in die *Closed List* ein.
4. Betrachte alle ausgehenden Kanten $\{e_0, \ldots, e_n\}$ von v. Sei v_i der Zielknoten von Kante e_i.
5. Falls v_i in der *Closed List* ist, betrachte nächsten adjazenten Knoten.
6. Falls $d(v) + c(e_i) < d(v_i)$:
   - Setze $d(v_i) = d(v) + c(e_i)$ und $p(v_i) = v$.
   - Falls v_i nicht in der Schlange enthalten ist, füge $(v_i, d(v_i) + h(v_i))$ in die Schlange hinzu
   - Ansonsten: Verringere den Schlüssel von v_i auf $d(v_i) + h(v_i)$.

Wie auch bei Dijkstra's Algorithmus nennt man eine Iteration dieses Verfahrens *Relaxierung* von v. Um den kürzesten Pfad nun zu extrahieren iteriert man wie bei Dijkstra vom Zielknoten aus immer die Vorgängerknoten entlang, bis man beim Startknoten angekommen ist.
Da $h(v) = 0\ \forall v \in V$ eine zulässige Heuristik ist, ist die Worst-Case-Laufzeit von A* die selbe wie die von Dijkstra's Algorithmus. Für eine perfekte Heuristik $h(v) = \mathcal{D}_w(v,t)$ ist der Suchraum von A* allerdings genau die Knoten des kürzesten Pfades. Das liegt daran, dass der Schlüssel jedes Knotens v in der Queue mindestens $\mathcal{D}_w(s,v) + h(v) = \mathcal{D}_w(s,v) + \mathcal{D}_w(v,t)$, also die Länge des kürzesten st-Pfades ist, der durch v geht. Für Knoten auf dem kürzesten Pfad ist dies minimal, somit werden die Knoten des kürzesten Pfades immer oben in der Queue liegen.
Für eine nicht perfekte, aber gute Heuristik wird der Suchraum im Vergleich zu Dijkstra auch stark eingeschränkt, da die Suche zielgerichtet ist, und nicht wahllos in jede Richtung gesucht wird.

2.4 Contraction Hierarchies [GSSD08]

Gegeben: Graph $G = (V,E)$ mit Kantengewichten $c : V \to \mathbb{N}_0$
Gesucht: Erweiterter Graph G^+ und Knotenhierarchie $h : V \to \mathbb{N}$ mit einigen Eigenschaften (siehe unten)



Ein Shortcut kann die Suche über eine Reihe von unwichtigen Knoten ersparen[1]

Contraction Hierarchies sind eine Vorverarbeitungstechnik, um die Suche von kürzesten Pfaden auf Straßennetzen erheblich zu beschleunigen. Hierbei wird ausgenutzt, dass Straßennetze hochgradig hierarchisch sind: Autobahnen sind schneller als Landstraßen, Landstraßen schneller als Hauptstraßen etc.
Contraction Hierarchies ist ein zweistufiger Algorithmus. In der ersten Stufe wird eine Hierarchie $H : V \to \mathbb{N}$ auf den Knoten des Graphen konstruiert. Im zweiten Schritt, die sogenannte *CH-Query*, können dann kürzeste Wege zwischen zwei beliebigen Knoten abgefragt werden. In der Query werden zwei Dijkstra-Suchen gestartet: Einmal vom Startknoten s und einmal vom Endknoten t. Dabei geht die Vorwärtssuche nur Kanten (u, v) ab, für die gilt: $H(v) > H(u)$. Es werden also nur Kanten betrachtet, die in der Hierarchie aufwärts gehen. Analog geht die Rückwärtssuche nur die Kanten (u, v) rückwärts ab, für die $H(u) > H(v)$ gilt. Die Suchen werden ausgeführt, bis ihre Warteschlangen leer sind. Unter allen Schnittknoten der beiden Suchräume wird dann der Knoten v mit kürzestem impliziten Pfad gesucht.
Ein *Auf-Ab-Pfad* ist ein Pfad, der so Partitioniert werden kann, dass der erste Teil des Pfades nur Aufwärtskanten hat, und der zweite Teil nur Abwärtskanten. Damit *Contraction Hierarchies* den kürzesten Pfad tatsächlich findet, muss sichergestellt werden, dass jeder kürzeste Pfad ein Auf-Ab-Pfad ist. Um das sicherzustellen, werden neue Kanten in den Graphen eingefügt. Diese Kanten nennen wir *Shortcuts*. Der Graph G mit hinzugefügten Shortcuts nennen wir *Erweiterter Graph* G^+. Das Tupel (G^+, H), also der erweiterte Graph mit Hierarchie, nennen wir *Contraction Hierarchy* (kurz: *CH*).

Baustein: Knotenkontraktion:
Die Knotenkontraktion von einem Knoten v modifiziert einen Graphen G wie folgt: Sei E^v_{in} alle eingehenden Kanten und E^v_{out} alle ausgehenden Kanten von v. Für jedes Tupel $((u, v), (v, w)) \in E^v_{in} \times E^v_{out}$ wird eine Kante (u, w) hinzugefügt, falls der kürzeste Pfad von u nach w über v führt. Die Kante (u, w) nennen wir *Shortcut*. Nachdem dies für jedes Tupel erfolgt ist, wird v aus dem Graphen gelöscht.

Schritt 1: Knotenordnung berechnen
Wir suchen eine Knotenordnung $(v_0, v_1, \ldots, v_N)$, sodass minimal viele Shortcuts erzeugt werden, wenn die Knoten in dieser Reihenfolge auf G kontrahiert werden. Da dieses Optimierungsproblem NP-Vollständig ist, nutzen wir eine Approximation:

1. Simuliere Knotenkontraktion für jeden Knoten $v \in V$, lösche also die Knoten nicht. Berechne für jede Kontraktion die Kantendifferenz, also die Anzahl der Shortcuts minus $grad(v)$.
2. Sortiere alle Knoten nach ihrer Kantendifferenz aufsteigend.

[1] Bildquelle: `https://commons.wikimedia.org/wiki/File:Shortcut_in_a_shortest_path.svg`, WhereAreMyPointersAt, CC BY-SA 4.0

3. Kontrahiere den Knoten mit minimaler Kantendifferenz. Berechne anschließend die Kantendifferenz für dessen Nachbarknoten neu und aktualisiere diese.
4. Gehe zum zweiten Schritt bis jeder Knoten kontrahiert wurde.

Die Reihenfolge der Kontraktionen beim obrigen Verfahren ist dann die gesuchte Knotenreihenfolge.

Schritt 2: Kontraktion und Hierarchie
Kontrahiere jeden Knoten in G in der berechneten Reihenfolge. Speichere dabei alle hinzugefügten Kanten in einer Liste. Setze $h(v)$ auf die Position von v in der Knotenreihenfolge. Für den zuerst kontrahierten Knoten v_0 gilt also $H(v_0) = 1$, für den zweiten Knoten v_1 dann $H(v_1) = 2$ usw.
Nachdem jeder Knoten kontrahiert wurde, füge alle erzeugten Shortcuts in den Originalgraphen ein. Diesen Graphen nennen wir den *erweiterten Graphen* G^+.

Schritt 3: Abfrage:
Um den kürzesten Weg von s nach t zu berechnen, werden zwei Dijkstra-Suchen gestartet: Die Vorwärtssuche mit Startknoten s und die Rückwärtssuche mit Startknoten t. Beide Suchen gehen nur in der Hierarchie nach oben, die Rückwärtssuche läuft die Kanten rückwärts entlang. Die Suchen werden vollständig ausgeführt. Wenn beide Suchen fertig sind, wird im Suchraumschnitt der beiden Suchen der Knoten v gesucht, für den $d = d_f(v) + d_b(v)$ minimal ist. d ist dann die Länge des kürzesten Pfades von s nach t.
Um den kürzesten Pfad zu erhalten, kann der implizite Pfad berechnet werden, indem vom Schnittknoten aus jeweils über die Parent-Knoten nach Vorne und Hinten traversiert wird. Anschließend müssen alle verwendeten Shortcuts *entpackt* werden, also durch ihre Kantenpaare ersetzt werden, die dieser Shortcut ersetzt hat. Dies muss rekursiv wiederholt werden, bis der Pfad keine Shortcuts mehr enthält. Damit das Entpacken möglich ist, muss bei der Knotenkontraktion für jeden Shortcut dessen zwei Ursprungskanten gespeichert werden.

Die Berechnung eines erweiterten Graphen und dessen Knotenhierarchie kann für hinreichend große Straßennetze bis zu einer Stunde dauern. Das liegt daran, dass jeder Knoten mindestens zweimal kontrahiert werden muss, und für jeden Knoten v $grad(v)$ viele Dijkstra-Anfragen gestartet werden müssen.
Wenn die Vorverarbeitung aber erledigt ist, ist der Zeitvorteil enorm: Während Dijkstra auf dem westeuropäischen Straßennetz mehrere Sekunden braucht um einen kürzesten Pfad z.B. von England nach Polen zu finden, benötigt die CH-Anfrage meist weniger als eine Millisekunde, allerhöchstens einige wenige Millisekunden. Das lässt sich dadurch erklären, dass die Suchräume der Vor- und Rückwärtssuchen winzig sind im Vergleich zum Suchraum von Dijkstra's Algorithmus.

2.4.1 Anmerkungen

- Da bei unserer Berechnungsweise der Knotenordnung der Graph schon in der richtigen Reihenfolge kontrahiert wird, ist Schritt 2 nicht mehr notwendig. Da sich die Knotenordnung während der Kontraktion ergibt, nennt man diese Approximation auch *lazy* oder *bottom-up*. Es gibt auch *top-down*-Approximationen, bei der die Knotenordnung vor der Kontraktion feststeht. Diese resultieren meistens in einer besseren Hierarchie und einem besseren erweiterten Graphen, aber auf Kosten von Vorverarbeitungszeit und Programmieraufwand.
- Bei der Kontraktion kann für jede Kante und jeden Shortcut geprüft werden, ob sie eine Aufwärts- oder Abwärtskante ist. So können Aufwärts- und Abwärtskanten jeweils in einen separaten Graph geschrieben werden, einen *Vorwärtsgraphen* und einen

Rückwärtsgraphen. Im Rückwärtsgraphen sind die Kanten invertiert. So muss man für die Abfrage seine Dijkstra-Implementation nicht für die Rückwärtstravesierung anpassen.

- Die Vorverarbeitung muss vollständig wiederholt werden, sobald sich der ursprüngliche Graph verändert. Falls also z.B. durch Stau eine Strecke langsamer wird, können *Contraction Hierarchies* das nicht darstellen ohne die gesamte Vorverarbeitung nochmal durchlaufen zu lassen.

2.5 BDV [ADGW10]

Eingabe: Ein Graph $G = (V, E)$, ein Startknoten s und ein Zielknoten t. Weiterhin drei Qualitätsparameter α, γ und ϵ
Ausgabe: Eine Menge von st-Pfaden mit Qualitätseigenschaften gemäß der Qualitätsparameter (siehe unten)

BDV verfolgt den Ansatz, alternative st-Pfade zu extrahieren, indem die kürzesten st-Pfade über einen Knoten v betrachtet werden. Diese Methode nennt man *Via-Knoten-Routing*, und der Knoten v ist der *Via-Knoten*.
Der kürzeste st-Pfad über einen Knoten v ist der Zusammenschluss aus dem kürzesten sv-Pfad und dem kürzesten vt-Pfad. Daher können die Via-Pfade gefunden werden, indem von s und von t aus jeweils eine Dijkstra-Suche gestartet wird. Die Suche von t aus läuft wieder die Kanten rückwärts ab. Wenn beide Suchräume aufeinandertreffen, impliziert der Schnittknoten einen Via-Pfad. Der Pfad kann dann wie im Abschnitt *Bidirektionaler A** beschrieben extrahiert werden. Wenn man die Suchen weiterlaufen lässt, werden immer mehr Via-Knoten gefunden. Das BD in BDV bedeutet *Bidirektionaler Dijkstra*
Sein $\mathcal{D}_v(s,t)$ die Länge des st-Via-Pfades über v. An die Via-Pfade werden mehrere Anforderungen gestellt:

1. *Limited Sharing*: Falls der Via-Pfad sich Kanten mit dem kürzesten st-Pfad teilt, darf der geteilte Anteil nicht länger als $\gamma \cdot \mathcal{D}_v(s,t)$ sein.
2. *Local Optimality*: Jede Teilstrecke von a nach b des Via-Pfades, die kürzer als $\alpha \cdot \mathcal{D}_v(s,t)$ ist, muss der kürzeste Pfad von a nach b sein.
3. *Uniformly Bounded Stretch (UBS)*: Jede Teilstrecke von a nach b des Via-Pfades, unabhängig seiner Länge, darf maximal die Länge $(1+\epsilon) \cdot \mathcal{D}(a,b)$ haben.

Aus der dritten Bedingung (UBS) folgt direkt, dass ein Via-Pfad maximal $(1+\epsilon)$ länger sein darf als der kürzeste st-Pfad. Daher können die beiden Dijkstra-Suchen abgebrochen werden, sobald der minimale Schlüssel in ihrer jeweiligen Queue größer als $(1+\epsilon) \cdot \mathcal{D}(s,t)$ ist.
Für alle Knoten v im Suchraumschnitt werden jetzt die implizierten st-Via-Pfade auf die drei Bedingungen geprüft und gefiltert. Während die *Limited Stretch*-Bedingung in Linearzeit geprüft werden kann, müssen für *Local Optimality* und *UBS* quadratisch viele kürzeste Distanzen ermittelt werden. Um dieses Problem zu umgehen, präsentieren Abraham und Goldberg einen Approximationsalgorithmus mit relativer Gütegarantie 2 für *UBS*. Diesen Test nennen sie *T-Test*:

Geben sei ein Parameter T. Sei a der Knoten auf dem sv-Teil des Via-Pfads, der am nächsten an v liegt, aber mindestens T Einheiten von v entfernt ist. Sei b Analog der Knoten auf dem vt-Teil des Via-Pfades, der den gleichen Bedingungen unterliegt. Wenn Teil des Via-Pfades von a nach b ein kürzester Pfad ist, dann besteht der Via-Pfad den

T-Test.

Im Paper zeigen die Authoren folgende drei Lemmas:

- Wenn ein Via-Pfad den T-Test besteht, sind alle Pfade der Länge T kürzeste Pfade.
- Wenn ein Via-Pfad den T-Test nicht besteht, gibt es einen Pfad der Länge $2T$ im Via-Pfad, der kein kürzester Pfad ist.
- Wenn ein Via-Pfad den T-Test mit $T = \beta \cdot \mathcal{D}(s,t)$ besteht und maximal $(1+\epsilon) \cdot \mathcal{D}(s,t)$ Einheiten lang ist, dann hat der Pfad einen maximalen UBS von $\frac{\beta}{\beta-\epsilon}$.

Daraus entsteht der BDV-Algorithmus:

1. Setze $\mathbb{P} = \varnothing$
2. Suche Via-Knoten durch die bidirektionale Dijkstrasuche wie oben beschrieben
3. Prüfe für jeden gefundenen Via-Pfad P_v:
   - P_v ist kürzer als $(1+\epsilon) \cdot \mathcal{D}(s,t)$
   - P_v besteht den *Limited Sharing*-Test
   - P_v besteht den T-Test mit $T = \alpha \cdot \mathcal{D}(s,t)$
4. Falls P_v alle Tests besteht und noch nicht gefunden wurde, füge P_v zur $\mathbb{P}$ hinzu.

Nachdem der Algorithmus abgeschlossen ist, beinhält $\mathbb{P}$ mehrere *st*-Via-Pfade, die den Qualitätskriterien approximiert entsprechen.

BDV ist für die Anwendung auf realen Straßengraphen zu langsam. Im Schnitt benötigt BDV auf dem westeuropäischen Straßennetzwerk circa eine halbe Minute, um alle Via-Pfade zu berechnen und zu filtern. In [ADGW13] wurden Techniken auf Basis von Reach und Contraction Hierarchies vorgestellt, um den Suchraum von BDV zu verringern. Diese können Alternativrouten in Echtzeit berechnen, nehmen dafür aber Einbußen von Qualität und Quantität der gefundenen Pfade in Kauf.

2.6 Penalty-Methode [KRS13]

Eingabe: Ein Graph G sowie Startknoten s und Zielknoten t.
Ausgabe: Ein dünn besetzter Alternativgraph $\mathcal{A}$ mit vielen guten Alternativrouten.

Die Penalty-Methode konstruiert einen Alternativgraphen, welcher sehr dünn besetzt ist aber dennoch viele gute Alternativrouten von s nach t beinhält. Da der Alternativgraph so dünn besetzt ist, kann in einer zweiten Phase dann ein Algorithmus zur Extraktion von Alternativrouten genutzt werden, der auf dem ursprünglichen Graph zu langsam wäre (z.B. X-BDV). Bader et al. stellen in [BDGS11] die Penalty-Methode vor. Der Alternativgraph $\mathcal{A} = (V, \varnothing)$ besitzt zu Anfang keine Kanten. Wir bezeichnen mit w die ursprüngliche Metrik aus der Eingabe und mit w' die sich in jeder Iteration verändernde Metrik. Eine Iteration der Penalty-Methode sieht wie folgt aus:

1. Finde den kürzesten Pfad P von s nach t
2. Addiere einen Bestrafungsfaktor pen_f auf alle Kanten von P: $w'(e) \leftarrow w'(e) + w(e) \cdot (1 + pen_f)\ \forall e \in P$

3. Addiere einen Rejoin-Bestrafungssummand pen_r auf alle Kanten, die nicht in P liegen aber zu Knoten von P adjazent sind:
   $w'(e) \leftarrow w'(e) + pen_r \; \forall e = (u, v) \notin P \wedge \exists w \in P \colon u = w \vee v = w$
4. Überprüfe den gefundenen Pfad auf seine Qualität
5. Falls die Qualitätskriterien erfüllt sind, füge die Kanten von P zu $\mathcal{A}$ hinzu.

Der kürzeste Pfad wird mit Dijkstra's Algorithmus ermittelt. Der Pfadbestrafungsfaktor pen_f wurde auf 4% gesetzt, der Rejoin-Faktor auf $pen_r = pen_f \cdot 0.002 \cdot \mathcal{D}_w(s, t)$. Die Kriterien an die Pfadqualität wurden nicht näher spezifiziert. In der Penalty-Methode von Bader et al. werden diese Iterationen 20 mal wiederholt. Auf den resultierenden Alternativgraphen kann dann BDV angewandt werden.
Anmerkung: In [KRS13] wird X-BDV genutzt, eine Variante von BDV aus [ADGW13] mit Pruning durch Contraction Hierarchies. Wir nutzen dieses Pruning nicht, da die Pfadextraktion viel schneller ist als die Berechnung des Alternativgraphen.
Da Dijkstra's Algorithmus auf größeren Straßengraphen mehr als eine Sekunde benötigt, um den kürzesten Pfad von s nach t zu finden, benötigt die gesamte Penalty-Methode von Bader et al. oft über 20 Sekunden um den Alternativgraphen zu berechnen. Kobitzsch et al. lösen dieses Problem in [KRS13], indem sie *Customizable Route Planning* (CRP) statt Dijkstra's Algorithmus zur Berechnung der kürzesten Pfade nutzen. Weiterhin ändern sie das Abbruchkriterium und die Pfadbestrafung. Sie spezifizieren außerdem konkrete Pfadauswahlkriterien für die gefundenen Pfade. Ihre Realisierung der Penalty-Methode sieht so aus:

- Kürzeste-Pfad-Suche: Customizable Route Planning (siehe unten)
- Abbruchbedingung: Länge des gefundenen Pfads auf Metrik w übersteigt $\varepsilon = 10\%$ der Länge des kürzesten Pfades
- Pfadbestrafung: Das Gewicht jeder Pfadkante wird mit $pen_f = 1.04$ multipliziert. Zusätzlich werden alle in den Pfad eingehenden Kanten additiv mit $pen_r = \alpha \cdot \sqrt{\mathcal{D}(s, t)}$ bestraft. Die Autoren empfehlen $\alpha = 0.5$
- Pfadauswahlkriterium: Ein Pfad muss mindestens eine Abweichung vom kürzesten Pfad mit einer gewissen Mindestlänge haben.

Durch die neue Pfadbestrafung sowie die neue Abbruchbedingung erreichen Kobitzsch et al. weitaus geringere Iterationszahlen als Bader et. al. Für weit entfernte Start-Ziel-Knotenpaare berichten die Autoren von einem Abbruch nach im Schnitt nur drei Iterationen. Trotz der geringen Iterationszahlen werden viele und qualitativ hochwertige Alternativpfade auf dem Alternativgraphen gefunden. Es bleibt zu klären, wie CRP funktioniert.
CRP ist eine metrikunabhängige Vorverarbeitungstechnik für Straßengraphen. Die Suche nach einem kürzesten Pfad erfolgt in drei Schritten:

1. Metrikunabhängige Vorverarbeitung
2. Anpassung der vorverarbeiteten Instanz auf die Metrik
3. Berechnug der kürzesten Route (Query).

Die Vorverarbeitung im ersten Schritt benötigt mehrere Minuten. Allerdings muss dieser Schritt nur einmal pro Graph ausgeführt werden. Die Anpassung auf eine gegebene Metrik benötigt in etwa eine Sekunde. Wenn die Instanz erst auf die Metrik angepasst wurde, kann die kürzeste Route innerhalb von wenigen Millisekunden berechnet werden.
Kobitzsch et al. nutzen aus, dass sich die Metrik in jeder Iteration nur auf ganz bestimmte

Weise ändert. Dadurch erreichen sie eine Verringerung der Laufzeit des zweiten Schritts auf weniger als 100 Millisekunden.
Durch die geringen Iterationszahlen in Kombination mit der schnellen Kürzeste-Wege-Suche können die Autoren von [KRS13] Laufzeiten von wenigen hundert Millisekunden selbst für die schwersten Testinstanzen erreichen. Somit ist die Penalty-Methode nun auch für interaktive Anwendungen brauchbar.

3. Unser Ansatz

Wir beschäftigen uns in dieser Arbeit mit der Penalty-Methode. Wie schon in Kapitel 3 beschrieben konstruiert die Penalty-Methode einen dünn besetzten Alternativgraphen, in welchem viele gute Alternativrouten zu finden sind. Dafür sucht sie mehrere Iterationen lang immer den kürzesten Pfad von Startknoten s zu Zielknoten t und erhöht dann die Kantengewichte der Pfadkanten. Jeder gefundene Pfad wird dann auf ein Gütekriterium geprüft und, falls dieses bestanden ist, zum Alternativgraphen hinzugefügt. Nach Abschluss der Penalty-Methode erhält man mithilfe von Via-Knoten-Algorithmen wie BDV dann viele gute Alternativpfade in kurzer Zeit. In Pseudocode sieht die Penalty-Methode so aus:

Algorithm 3.1: Penalty-Methode

```
A ← (V, ∅)
P ←berechne_kürzesten_weg(G, s, t)
while !abbruchbedingung(...) do
  bestrafe_pfad(G, P)
  P ← berechne_kürzesten_weg(G, s, t)
  if pfad_akzeptabel(P) then
    A ← A ∪ P
  end if
end while
```

Wir bezeichnen ab jetzt die originale Metrik von G mit w und die Metrik nach Pfadbestrafung mit w'. Der berechnete Alternativgraph heißt $\mathcal{A}$.
Die Funktionen `berechne_kürzesten_weg`, `abbruchbedingung`, `bestrafe_pfad` und `pfad_akzeptabel` müssen noch definiert werden. Die Qualität des resultierenden Alternativgraphen wird nicht von `berechne_kürzesten_weg` beeinflusst (solange die Funktion korrekt ist). Unter der Annahme dass der Hauptrechenaufwand der Penalty-Methode in `berechne_kürzesten_weg` liegt, beeinflusst `pfad_akzeptabel` die Laufzeit nicht maßgeblich.
Wie in Kapitel 3 beschrieben stellten Kobitzsch et al. in [KRS13] eine Realisierung der Penalty-Methode für den interaktiven Gebrauch vor. Wir betrachten und evaluieren in dieser Arbeit eine alternative Realisierung der Penalty-Methode.
Realisierung 1:

- Kürzeste-Pfad-Suche: Bidirektionaler A*-Algorithmus mit CH-Potentialen als Heuristik
- Abbruchbedingung: Länge des gefundenen Pfads auf Metrik w übersteigt $\varepsilon = 10\%$ der Länge des kürzesten Pfades
- Pfadbestrafung: Das Gewicht jeder Pfadkante wird mit $pen_f = 1.04$ multipliziert. Zusätzlich werden alle in den Pfad eingehenden Kanten additiv mit $pen_r = \alpha \cdot \sqrt{\mathcal{D}(s,t)}$ bestraft. Wir setzen $\alpha = 0.5$
- Pfadauswahlkriterium: Ein Pfad muss mindestens eine Abweichung vom kürzesten Pfad mit einer gewissen Mindestlänge haben.

Bis auf die Kürzeste-Pfad-Suche stimmt unsere Realisierung mit der von Kobitzsch et al. überein. Dadurch erhoffen wir, die geringen Iterationszahlen sowie die guten Qualitätswerte aus [KRS13] reproduzieren zu können. Durch die Nutzung des A*-Algorithmus für die Kürzeste-Wege-Suche verringern wir den Implementationsaufwand der Penalty-Methode. Weiterhin müssen wir keine Anpassung an die neue Metrik vornehmen, da eine Heuristik für den A*-Algorithmus auch noch gültig ist, wenn die Werte der Metrik erhöht werden. Wir erhalten eine Heuristik für den A*-Algorithmus bezüglich der Metrik w durch den CH-Potentiale-Algorithmus. Zusätzlich evaluieren wir noch zwei weitere Realisierungen der Penalty-Methode, welche beide A* mit CH-Potentialen nutzen:
Realisierung 2:

- Abbruchbedingung: Länge des gefundenen Pfads auf Metrik w' übersteigt $\varepsilon = 10\%$ der Länge des kürzesten Pfades
- Rest: Wie in Realisierung 1

Realisierung 3:

- Abbruchbedingung: Länge des gefundenen Pfads auf Metrik w' übersteigt $\varepsilon = 10\%$ der Länge des kürzesten Pfades
- Pfadbestrafung: In jeder Iteration wird das Gewicht jeder Kante von $\mathcal{A}$ mit $pen_f = 1.04$ multipliziert. Die Pfadbestrafung erfolgt nach Hinzufügen eines Pfades zum Graphen.
- Rest: Wie in Realisierung 1

Die beiden Varianten sind durch die Evaluationsergebnisse von Realisierung 1 motiviert. In Realisierung 2 erhoffen wir uns durch das neue Abbruchkriterium eine geringere Anzahl von Iterationen bei ähnlicher Erfolgsquote. In Realisierung 3 erwarten wir noch weniger Iterationen, da durch die Bestrafung des gesamten Alternativgraphen ein "Zurückspringen" auf schon gefundene Pfade unwahrscheinlich wird. Natürlich folgt aus der geringeren Anzahl an Iterationen, dass weniger gute Alternativpfade gefunden werden können. Wir evaluieren die Laufzeit sowie die Qualitätseinbüßungen der verschiedenen Realisierungen in Kapitel 4.

CH-Potentiale steht für *Contraction Hierarchy-Potentiale* und ist ein Algorithmus, der für eine zur Vorberechnungszeit bekannte Metrik w eine perfekte Heuristik $h(v) = \mathcal{D}_w(v,t)$ berechnen kann. Dafür wird eine Contraction Hierarchy benötigt. Mithilfe einer solchen Heuristik lässt sich der kürzeste *st*-Pfad in Millisekunden mithilfe von A* finden. Allerdings ist die Heuristik nach Anwendung von Penalties nicht mehr akkurat, sodass in jeder Iteration der Suchraum des A*-Algorithmus wächst. Wir untersuchen in dieser Arbeit ob die Änderungen an der Metrik gering genug sind um gute Laufzeiten für jede Iteration zu erhalten. Um das Wachstum des Suchraums zu beschränken und die Leistung von Mehrkernprozessoren auszunutzen nutzen wir einen bidirektionalen A*-Algorithmus, welcher auf zwei CPU-Kernen läuft.

Die Performance des A*-Algorithmus hängt von der Qualität seiner Heuristik ab. Sei w die Originalmetrik und w' die Metrik mit Kantenbestrafungen von G. Für die Heuristik $h^*(v) = \mathcal{D}_w(v, t)$ ist der Suchraum von A* gerade die Knoten des kürzesten st-Pfades für jeden Startknoten s. Je näher eine Heuristik h an der perfekten Heuristik h^* liegt, desto geringer ist der Suchraum.
In [SZ21] wurden *CH-Potentiale* vorgestellt. Mithilfe von CH-Potentialen kann eine perfekte Heuristik für den A*-Algorithmus bezüglich der Originalmetrik w *lazy* berechnet werden. Das bedeutet, dass die Heuristik eines Knotens erst dann berechnet wird, wenn sie in der A*-Suche tatsächlich benötigt wird.

CH-Potentiale [SZ21]

Eingabe: Ein Graph $G = (V, E)$, eine CH für G und ein Zielknoten t. **Ausgabe:** Eine perfekte Heuristik für den A*-Algorithmus auf G mit Zielknoten t.

Für jeden Knoten wird eine tentative Distanz $d : V \to \mathbb{N}_0$ und ein Potential $p : V \to \mathbb{N}_0$ gespeichert. Am Anfang ist die tentative Distanz jedes Knotens ∞ und jedes Potential $\perp$.

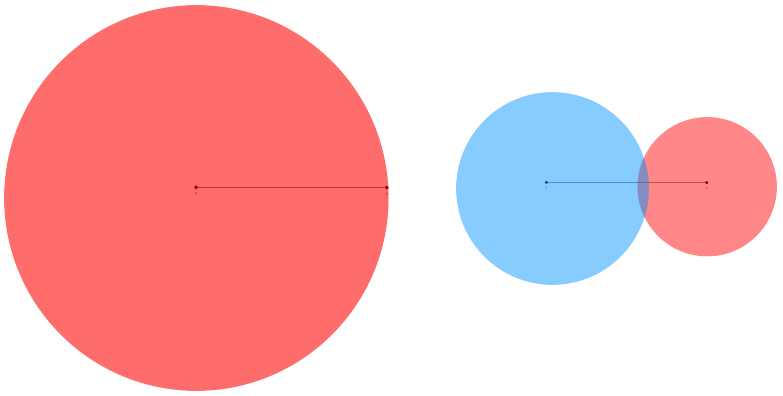
1. Führe eine Rückwärtssuche wie bei der Query-Phase von Contraction Hierarchies vom Zielknoten aus. Lasse diese laufen bis die Queue leer ist. Für jeden Knoten v im Suchraum der Rückwärtssuche: Setze $d(v)$ und $p(v)$ auf die gefundene Distanz.
2. Bei Abfrage der Heuristik eines beliebigen Knotens v: Falls $p(v) = \perp$:
   a) Setze $p(v) = d(v)$
   b) Betrachte alle Aufwärtskanten $e = vw$ in der CH. Setze $p(v) = \min(p(v), p(w) + c(e))$. Beachte, dass $p(w)$ ein rekursiver Aufruf dieses Schritts ist.

   Gebe dann $p(v)$ zurück.

Der zweite Schritt wird dabei *lazy* ausgeführt: Erst wenn die A*-Suche einen Knoten das erste mal betrachtet wird die Heuristik des Knotens berechnet. Da mit einer perfekten Heuristik der Suchraum sehr klein ist, muss die Heuristik nur für sehr wenige Knoten berechnet werden. Da Contraction Hierarchies als Beschleunigungstechnik genutzt wird, benötigt die Berechnung der Heuristik für einen einzelnen Knoten wenige hundert Mikrosekunden.

Unsere Arbeit ist teilweise durch die Veröffentlichung des CH-Potentiale-Algorithmus motiviert. Die Implementation der CH-Potentiale erweist sich als recht einfach, die erreichte Laufzeit ist jedenfalls für wenige Iterationen sehr gut. In [SZ21] wurde untersucht, wie stark sich der Suchraum einer A*-Suche mit CH-Potentialen vergrößert, wenn die Metrik des Graphen mit dem Faktor $1 + \delta$ multipliziert wird. Für $\delta = 4\%$ vergrößert sich der Suchraum, und damit auch die Laufzeit, ungefähr um den Faktor 8. Es wird sich zeigen ob der Suchraum in der Penalty-Methode pro Iteration stärker oder schwächer wächst als in [SZ21] angegeben. Für ein geringeres Wachstum spricht, dass nicht die gesamte Metrik verändert wird, sondern nur ein kleiner Teil. Für ein stärkeres Wachstum spricht die Rejoin-Penalty, welche im allgemeinen mehr Kanten betrifft als die Pfad-Penalty und meist eine Kante stärker als um 4% bestaft.
Der bidirektionale A*-Algorithmus verringert den Suchraum der A*-Suche. Man stelle sich den Suchraum als Kreis vor, welcher um den Startknoten herum wächst[1]. Die Fläche dieses Kreises repräsentiert den Suchraum des A*. Beim einfachen A*-Algorithmus wäre der Suchraum also symbolisch $\pi \cdot r^2$, wobei der Radius r des Suchraums die Anzahl der

[1] Das ist so nicht ganz richtig. Mit einer halbwegs sinnvollen Heuristik ist der Suchraum eher ein Wassertropfen, dessen Spitze am Zielknoten ist. Dennoch hilft das Bild dem Verständnis.



(a) Einfach (b) Bidirektional

Abbildung 3.1: Vergleich von Suchräumen

Knoten im kürzesten *st*-Pfad wäre. Für eine bidirektionale A*-Suche wäre der Suchraum dagegen repräsentiert durch zwei Kreise, welche ihren Ursprung in jeweils dem Start- und Zielknoten haben. Deren Fläche berechnet sich durch $\pi \cdot (\frac{r}{2})^2 = \frac{1}{4}\pi r^2$. Da es zwei Suchkreise gibt, wird der Suchraum also halbiert. Da der bidirektionale A* weiterhin echt parallel auf zwei Prozessorkernen laufen kann, kann so nochmal ein Speedup von Faktor 2 erreicht werden.

Bidirektionaler A*-Algorithmus [GW05]
Statt einer A*-Suche werden im bidirektionalen A*-Algorithmus zwei A*-Suchen gestartet, einmal vom Startknoten und einmal vom Zielknoten aus. Die Suche vom Startknoten nennt man *Vorwärtssuche* und sie erhält eine Heuristik mit dem Zielknoten als Ziel. Analog heißt die Suche vom Zielknoten aus *Rückwärtssuche* und erhält eine Heuristik mit dem Startknoten als Ziel. Zusätzlich wird eine tentative Distanz μ und ein bester Knoten v^* gespeichert. Diese repräsentieren den bislang kürzesten gefundenen Pfad. Zu Anfang der bidirektionalen Suche gilt $\mu = \infty$ und $v^* = \perp$.
Die Suchen funktionieren wie in Kapitel 4 beschrieben, aber mit einem Zusatz: Wenn die Vorwärtssuche eine Kante (u, v) relaxiert, wird geprüft, ob v in der Closed-List der Rückwärtssuche enthalten ist. Falls dem so ist, wird geprüft, ob der kürzeste Pfad durch v kürzer ist als der bislang gefundene kürzeste Pfad. Die Länge des kürzesten Pfades durch v ist $d_f(u) + w(uv) + d_r(v)$, wobei d_f die Distanzfunktion der Vorwärtssuche und d_r die Distanzfunktion der Rückwärtssuche ist. Falls diese Länge kleiner als μ ist, wurde ein neuer kürzester Pfad gefunden. Setze $\mu = d_f(u) + w(uv) + d_r(v)$ und $v^* = v$. Dieser Schritt wird Analog auch in der Rückwärtssuche gemacht.
Eine A*-Suche kann auch als eine Dijkstra-Suche mit modifizierter Metrik betrachtet werden. Eine A*-Suche mit Metrik w und Heuristik h ist äquivalent zu einer Dijkstra-Suche mit Metrik $w'(uv) = w(uv) + h(v) - h(u)$. Wenn also die Vorwärts- und Rückwärtssuche unterschiedliche Metriken haben, suchen die korrespondierenden Dijkstra-Suchen auf unterschiedlichen Graphen. Dadurch ist es schwierig, ein sinnvolles Abbruchkriterium für die beiden Suchen zu finden, da die Suchräume wenige Gemeinsamkeiten aufweisen können.
Um dieses Problem zu lösen wird eine neue Heuristikfunktion für Vorwärts- und Rückwärtssuche eingeführt, die ein Mittel zwischen den beiden Heuristiken bildet. Sei π_f die originale Vorwärtsheuristik und π_r die originale Rückwärtsheuristik.

- $p_f(v) = \frac{\pi_f(v) - \pi_r(v) + \pi_r(t)}{2}$
- $p_r(v) = \frac{\pi_r(v) - \pi_f(v) + \pi_f(s)}{2}$

Für diese Heuristikfunktionen gilt nun $p_f(v)+p_r(v)$ =const. $\forall v \in V$ und weiterhin $p_f(t) = 0$ und $p_r(s) = 0$, wie man es von einer Heuristik erwarten würde. Nun suchen die korrespondierenden Dijkstra-Suchen auf Graphen mit gleicher Metrik.
Die Suchen brechen dann ab, wenn $k_f^* + k_r^* \geq \mu + p_f(s)$, wobei k_f^* der oberste Queueeintrag der Vorwärtssuche, k_r^* der oberste Queueeintrag für die Rückwärtssuche und $p_f(s)$ die Heuristik der Vorwärtssuche vom Startknoten ist. $k_f^* + k_r^* = d(v_f^*) + d(v_r^*) + p_f(v_f^*) + p_r(v_f^*)$ ist eine untere Schranke für die zweifache Länge jedes Pfades, der noch gefunden werden kann, da die Heuristik die tatsächliche Distanz zum Zielknoten überschätzt. $\mu + p_f(s)$ ist eine obere Schranke der doppelten Länge des kürzesten Pfades. Wenn die Ungleichung also erfüllt ist, kann kein kürzerer Pfad mehr gefunden werden.
Die Suchen werden abwechselnd oder parallel ausgeführt. Tatsächlich liefert der Algorithmus für jede Wechselstrategie die korrekte Lösung. Goldberg et al. führen immer einen Schritt der Suche aus, die den kleineren obersten Schlüssel in ihrer Queue hat. Nach Abbruch der Suchen kann die kürzeste Route extrahiert werden, indem die kürzeste sv^*- und v^*t-Route berechnet wird und beide Routen zusammengefügt werden. Die Länge der kürzesten Route steht in μ.

Da jede Wechselstrategie zu einem korrekten Ergebnis führt, lassen wir die Vorwärts- und Rückwärtssuche echt parallel auf zwei verschiedenen Kernen laufen. Wenn man jedem Thread seine eigene Heuristikinstanz gibt und μ sowie v^* nur durch die Vorwärtssuche aktualisiert wird, ist keine Synchronisation zwischen den Threads notwendig. Die separaten Heuristikinstanzen sind notwendig, da während der Berechnung der CH-Potentiale falsche Heuristiken in der Potentialfunktion stehen können. Durch die parallele Ausführung erreichen wir einen Speedup von Faktor 2 gegenüber der pseudoparallelen Ausführung.

3.1 Implementationsdetails

Zur Berechnung von Alternativrouten von s nach t auf einem Graphen G werden die folgenden Schritte ausgeführt:

1. Berechnung einer Contraction Hierarchy für G
2. Initialisierung von CH-Potentialen auf der CH für G
3. Berechnung des Alternativgraphen $\mathcal{A}$ mithilfe der Penalty-Methode und bidirektionalem Astar mit CH-Potentialen.
4. Berechnung von Alternativrouten mithilfe von BDV.
5. Evaluation der Alternativrouten nach Qualitätsparametern wie in Kapitel 2 definiert.

3.1.1 Berechnung der CH

Ausschlaggebend für die Qualität einer Contraction Hierarchy ist die Reihenfolge, in welcher die Knoten kontrahiert werden. Wir nutzen eine *Bottom-Up*-Knotenordnung, bei der der als nächstes zu kontrahierende Knoten erst während der Kontraktion des Graphens berechnet wird.
Sei $G \setminus \{v\}$ der Graph, der aus der Knotenkontraktion von v auf G resultiert. Die Kantendifferenz Δ_v von v ist definiert als die Anzahl der Kanten von $G \setminus \{v\}$ minus die Anzahl der Kanten von G. Wir berechnen die Kantendifferenz Δ_v jedes Knotens v in G indem wir die Kontraktion jedes Knotens simulieren. Wir speichern (v, Δ_v) in eine Liste $\mathcal{L}$ und sortieren diese Liste aufsteigend nach der Kantendifferenz.
Nachdem die Kantendifferenz jedes Knotens berechnet wurde, beginnt die tatsächliche Kontraktion. Wir kontrahieren den Knoten v^* mit der kleinsten Kantendifferenz in $\mathcal{L}$ und

löschen dessen Eintrag aus $\mathcal{L}$. Jeder zu G hinzugefügte Shortcut uv wird in einer Liste $\mathcal{S}$ gespeichert. Weiterhin speichern wir v^* in einer Liste $\mathcal{H}$. Danach wird für jeden Nachbarknoten von v^* die Kantendifferenz neu berechnet und deren Einträge in $\mathcal{L}$ aktualisiert. Danach wird der neue Knoten mit minimaler Kantendifferenz in $\mathcal{L}$ gesucht und kontrahiert. Dieses Verfahren wird ausgeführt bis $\mathcal{L}$ leer ist.
Nun augmentieren wir G mit den Kanten in $\mathcal{S}$ und erhalten den Graphen G^+ der Contraction Hierarchy. Die Hierarchie der Knoten ergibt sich aus der Reihenfolge der Knoten in $\mathcal{O}$, $h(v)$ ist der Index von v in $\mathcal{H}$. Wir speichern G^+ nicht als einen, sondern als zwei Graphen: Ein Graph für alle *Aufwärtskanten* (uv, $h(u) < h(v)$) und ein Graph für alle *Abwärtskanten*. Der Graph mit den Abwärtskanten speichert jede Kante falschrum (vu statt uv). Somit muss eine Dijkstra-Implementation nicht für die Traversierung von Rückwärtskanten modifiziert werden.
Dijkstra's Algorithmus:
Für die Zeugensuche in der Kontraktion wird Dijkstra's Algorithmus genutzt. Da jede Zeugensuche nur einen sehr kleinen Suchraum hat, ist die Performance ausreichend. Allerdings ist das Speichern der Distanzen jedes Knotens problematisch: Eine normale Liste der Größe $|V|$ funktioniert nicht, da nach Abschluss einer Dijkstra-Suche die Liste geleert werden muss. Dies benötigt $\mathcal{O}(|V|)$ und $|V|$ ist sehr groß. Eine Möglichkeit wäre die Nutzung einer Hashmap, damit nur die Werte von Knoten gespeichert werden, die tatsächlich besucht werden, aber eine Hashmap hat einen zu großen konstanten Overhead. Wir konstruieren zur Lösung dieses Problems die Datenstruktur `TimestampVector`:
Ein `TimestampVector` beinhält zwei Arrays der Größe $|V|$: Ein Wertevektor `value` und einen Zeitvektor `time`. Weiterhin wird ein Zeitstempel `t` und ein Standardwert `default` gespeichert. Ein `TimestampVector` hat drei Funktionen:

- `get(index)`: Falls `time[index] == t`, dann wird `value[index]` zurückgegeben. Ansonsten `default`
- `set(index, v)`: Setzt `value[index] = v` und `time[index] = t`.
- `clear()`: Erhöht `t` um 1.

Alle drei Funktionen laufen in $\mathcal{O}(1)$. Somit muss ein `TimestampVector` nur einmal in $\mathcal{O}(n)$ initialisiert werden und kann dann durchgehend für Dijkstra's Algorithmus verwendet werden. Wir nutzen `TimestampVector` auch für bidirektionalen A* sowie für die Potentialvektor von CH-Potentialen.
CH-Ordnung:
Ein zweites Performanceproblem ist die Berechnung des minimalen Elements in $\mathcal{L}$. Für eine normale Liste läuft dies in $\mathcal{O}(n)$ und ist, da $\mathcal{L}$ sehr groß ist, zu langsam. Auch wenn ein binärer Heap, welchen wir schon für Dijkstra's Algorithmus nutzen, genutzt wird, erweist sich die Performance als mangelhaft, da die `decrease-key`-Operation trotz Laufzeit $\mathcal{O}(\log n)$ zu lange braucht.
Zur Lösung dieses Problems nutzen wir eine `BucketQueue`. Ausschlaggebend dafür ist, dass es nur wenige Schlüsselwerte in $\mathcal{L}$ gibt, denn die Knotendifferenz ist eine Ganzzahl und liegt meist zwischen -10 und 100.
Ein *Bucket* ist eine Liste von Knoten. Um eine `BucketQueue` zu initialisieren, erstellen wir einen Bucket für jede Kantendifferenz in $\mathcal{L}$ und fügen jeden Knoten in seinen entsprechenden Bucket ein. Weiterhin speichern wir noch zwei Arrays `bucket` und `index` der Größe $|V|$. `bucket[v]` gibt den Bucket des Knotens v (also dessen Kantendifferenz Δ_v) an und `index[v]` die Position von v in seinem Bucket. Eine `BucketQueue` stellt nun folgende Funktionen bereit:

- `pop()`: Löscht den obersten Knoten aus dem Bucket für die geringste Kantendifferenz mithilfe eines Swap-Erase (siehe unten) und gibt diesen Knoten zurück. Es wird `bucket[v] = -1` und `index[v] = -1` gesetzt.
- `push(v, delta_v)`: Fügt v in den Bucket für die Kantendifferenz Δ_v ein. Falls dieser Bucket nicht existiert, wird er erstellt. Die Werte von `bucket[v]` und `index[v]` werden entsprechend aktualisiert.
- `change_key(v, delta_v)`: Löscht v aus seinem Bucket mithilfe von Swap-Erase. Anschließend wird `push(v, delta_v)` ausgeführt.

Bei einem Swap-Erase von v wird der Eintrag von v mit dem letzten Knoten des Buckets getauscht. Anschließend wird der letzte Eintrag des Buckets gelöscht. Dieses Löschverfahren hat konstante Laufzeit, behält aber die Ordnung der Liste nicht bei. Für die Buckets ist das allerdings unerheblich. Durch die Ausnutzung von Swap-Erase hat jede Operation von `BucketQueue` konstante Laufzeit.
Durch die `BucketQueue` konnte die Berechnung einer CH für ein westeuropäischen Straßengraphen mit $|V| \approx 10^7$ und $|E| \approx 2 \cdot 10^7$ in circa 10 Minuten erfolgen (auf einem AMD Ryzen 3600 mit DDR4-RAM). Zum Vergleich: Mit einem binären Heap hätte die Berechnung auf denselbem Graphen eine knappe Stunde gebraucht.
Die gewählte Knotenordnung ist schnell und einfach zu berechnen, aber nicht optimal. Die Berechnung der perfekten Knotenordnung ist NP-Vollständig [BDS+08], allerdings lassen sich mit anderen Verfahren bessere Ergebnisse erzielen. Daher nutzen wir für die Auswertung unseres Algorithmus nicht unsere eigene, sondern eine von dem Programm *RoutingKit*[2] generierte Contraction Hierarchy.

3.1.2 Implementierung von CH-Potentialen

Die effiziente Implementierung von CH-Potentialen gestaltet sich als sehr einfach. Wir nutzen einen `TimestampVector` um die Potentiale zu speichern. Ansonsten lässt sich der Pseudocode aus [SZ21] fast eins-zu-eins in C++-Code übersetzen. Da wir später bei der bidirektionalen A*-Suche jedem Thread seine eigenen Potentiale geben, benötigen wir auch keine Synchronisationsstrukturen.

3.1.3 Penalty-Methode und bidirektionaler A*

Auch für die Penalty-Methode lässt sich der weiter oben angegebene Pseudocode eins-zu-eins in C++-Code übersetzen. Die Vorwärts- und Rückwärtssuchen des bidirektionalen A* sind äquivalent zu zwei Dijkstra-Suchen, mit der kleinen Änderung dass die Einträge in der Queue nicht die tentative Distanz des Knotens als Schlüssel haben, sondern die tentative Distanz des Knotens plus die Heuristik des Knotens. Es werden also dieselben Datenstrukturen für den bidirektionalen A* verwendet, wie sie schon für Dijkstra's Algorithmus genutzt werden. Die Abbruchbedingung und das Pruning-Verfahren kann dann direkt aus [GW00] übernommen werden. Um die beiden Threads der Vorwärts- und Rückwärtssuche nicht synchronisieren zu müssen, prüft lediglich die Vorwärtssuche, ob der Zielknoten einer zu relaxierenden Kante schon von der Rückwärtssuche besucht wurde. Weiterhin erhält jeder Thread seine eigene CH-Potentiale-Instanz.
Die Laufzeit der Berechnung, ob ein Knoten in einer Liste enthalten ist, ist linear in der Länge der Liste. Daher ist es nicht ratsam, den Suchraum als normale Liste zu speichern. Speichern als `TimestampVector` mit boolschen Werten ist möglich, benötigt aber viel mehr Speicherplatz als eigentlich notwendig ist, da für jeden `bool`-Eintrag (1 Bit) auch noch der Zeiteintrag (32 bit) gespeichert werden muss. Speichern als `HashSet` ist auch möglich, aber je nach Implementierung der Hashfunktion recht langsam. Wir konstruieren

[2] `https://github.com/RoutingKit/RoutingKit`

die Datenstruktur `BoolSet` zur Lösung des Problems.
Ein `BoolSet` beinhält ein Boolean-Array `has` der Größe $|V|$ sowie eine Liste `delete_list`, welche zu Anfang leer ist. Ein `BoolSet` stellt folgende Funktionen bereit:

- `has(v)`: Gibt `has[v]` zurück.
- `set(v)`: Setzt `has[v] = true` und fügt v zur `delete_list` hinzu.
- `clear()`: Iteriert durch alle Einträge von `delete_list` und setzt deren `has`-Wert auf `false`.

`has` hat somit eine konstante Laufzeit. Die Laufzeit von `clear` ist Linear in der Suchraumgröße der A*-Suche. Für kleine Suchräume ist `clear` also sehr schnell und für große Suchräume ist die Laufzeit von `clear` vernachlässigbar klein im Vergleich zur A*-Suche selbst.

3.2 Berechnung der Qualitätsmerkmale

Die Berechnung des Uniformly Bounded Stretch und der Local Optimality benötigt quadratisch viele Kürzeste-Wege-Anfragen (abhängig zur Anzahl Knoten im Pfad), und ist daher entsprechend langsam. Wir nutzen CH-Potentiale, um die Berechnung zu beschleunigen, hier Beispielhaft für Local Optimality:
Anstelle für jedes `i`-`j`-Paar also eine Shortest-Path-Anfrage mit einer CH-Query zu starten,

Algorithm 3.2: Local Optimality-Berechnung

```
ret = P.length
ch_pot = CHPotentials(g)
for i = 1; i < P.num_nodes; i++ do
  ch_pot.set_target(P[i])
  for j = 0; j < i; j++ do
    optimal_dist = ch_pot.get(P[j])
    path_dist = get_dist_on_path(P[i], P[j])
    if optimal_dist < path_dist and path_dist < ret then
      ret = path_dist
    end if
  end for
end for
return ret
```

nutzen wir die schon berechneten Distanzen für Zielknoten `P[i]` für weitere Iterationen der inneren For-Schleife. Im Vergleich zu einer seperaten CH-Query für jedes Knotenpaar können wir einen Speedup von etwa Faktor 5 zur Berechnung der Qualitätswerte erreichen. Dadurch konnten unsere Testinstanzen schneller evaluiert werden und entsprechend mehr Tests ausgeführt werden.

4. Ergebnisse

4.1 Setup

Die Ergebnisse wurden auf einem Intel Xeon E5-1630 mit einem verfügbaren Hauptspeicher von 128GB berechnet. Alle Experimente liefen auf dem westeuropäischen Straßengraphen von der neunten DIMACS-Implementationschallenge ($|V| \approx 18 \cdot 10^6$, $|E| \approx 21 \cdot 10^6$). Als Metrik wird die Reisezeit zwischen zwei Knoten in Zehntelsekunden gewählt. Der Graph ist gerichtet und stark zusammenhängend. Für diesen Graphen wurden maximal 16GB des verfügbaren Speichers verwendet. Da unsere Implementierung nur auf Laufzeit, nicht auf Speicherverbrauch optimiert wurde, können die folgenden Ergebnisse wahrscheinlich mit geringerem Speicherverbrauch produziert werden.
Das Programm wurde in C++ geschrieben und mit der *GNU-Compiler-Collection*(`gcc`) kompiliert. Es wird mit den Flags `-O3 -pthread -std=c++17` kompiliert. Wir nutzen weiterhin die Header-Only-Bibliotheken *aixlog* (zum Loggen), *cxxopts* (zum Parsen von Kommandozeilenparametern) und *spatium* (Zum Zeichnen von Bildern ins PPM-Format). *aixlog* und *cxxopts* stehen unter MIT-Lizenz, *spatiumLib* unter GNU-GPL3-Lizenz.
Wir führen für jede Realisierung der Penalty-Methode drei Testläufe durch:

1. 1000 zufällige Start-Ziel-Knotenpaare.
2. 100 zufällige Start-Ziel-Knotenpaare mit variierenden Parametern α, ε und pen_f.
3. 1000 zufällige Startknoten, für jeden Startknoten alle Knoten mit Dijkstra-Rank 2^k vom Startknoten aus als Zielknoten, wobei $k \in \mathbb{N}$, $k \geq 10$.

Der Knoten mit Dijkstra-Rank n bezüglich eines Knotens s ist der Knoten, den eine Dijkstra-Suche mit Quellknoten s als n-tes besucht. Je höher der Dijkstra-Rank, desto länger ist der Pfad zu diesem Knoten. Wir vergleichen alle Ergebnisse mit den Ergebnissen aus [KRS13] und mit den Ergebnissen von X-BDV, X-CHV und X-REV aus [ADGW10]. Weiterhin evaluieren wir die gefundenen Alternativpfade nach ihren Qualitätsparametern *UBS*, *LO* und *Sharing* wie in Kapitel 3 definiert. Für jede Testinstanz werden die gefundenen Pfade nach einer Bewertungsfunktion $f(P) = 2 \cdot len_w(P) + \sigma(P) - pl(P)$ aufsteigend sortiert (Entnommen aus [ADGW10]). $\sigma(P)$ gibt den Sharing-Wert von P an, $pl(P)$ die Länge des längsten Teilpfades von P, der in beiden BDV-Suchbäumen enthalten ist. Die Bewertungsfunktion gibt ein Maß für die Qualität eines Pfades an. Es wird dann geprüft, mit welcher relativen Häufigkeit ein k-ter Alternativpfad ($k \in \{1, 2, 3\}$) gefunden wurde. Außerdem werden die durchschnittlichen Qualitätsparameter für den k-ten Alternativpfad

bestimmt.
Zur Visualisierung einiger Daten nutzen wir *Boxplots* (siehe Abb. 2.1). Das Gebilde über einem Wert der x-Achse nennt man einen *Boxplot*. Die horizontale Linie in einem Boxplot gibt dem Median der Werte an. Das Rechteck spannt vom 25%-Perzentil bis zum 75%-Perzentil. Die Linien, die aus den Kästen hervorgehen, nennt man *Whisker* und erstrecken sich um den 1.5-fachen Quartilsabstand nach oben bzw. nach unten. Werte außerhalb der Whisker sind Außreißer und werden als hohle Kreise markiert.

4.2 Realisierung 1: Originale Penaltyparameter

Zunächst evaluieren wir die Penalty-Methode mit den Parametern, wie sie in [KRS13] gegeben sind. Kobitzsch et al. berichten von gerade mal zwei bis vier Iterationen für Testinstanzen mit den Dijkstra-Ranks 2^{23} und 2^{24}. Durch diese geringen Iterationszahlen und Ausnutzung von CRP erreichen sie auch für hohe Dijkstra-Ranks schnelle Laufzeiten von weit unter einer Sekunde. Da wir nur die Berechnung der kürzesten Route verändert haben, erwarten wir ähnliche Iterationszahlen für die einzelnen Testfälle.

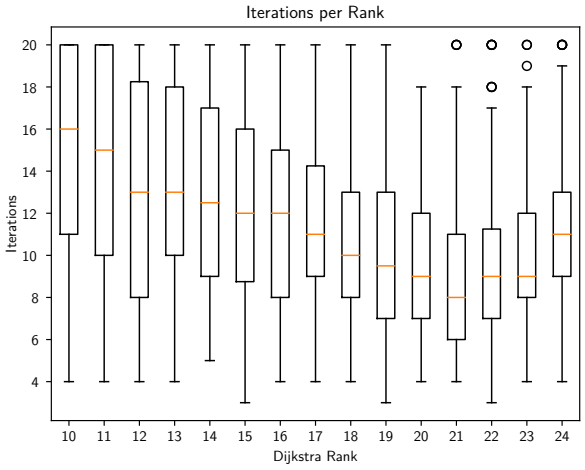


Abb. 2.1: Anzahl der Iterationen in Abhängigkeit zum Dijkstra-Rank. Die x-Achse ist logarithmisch.

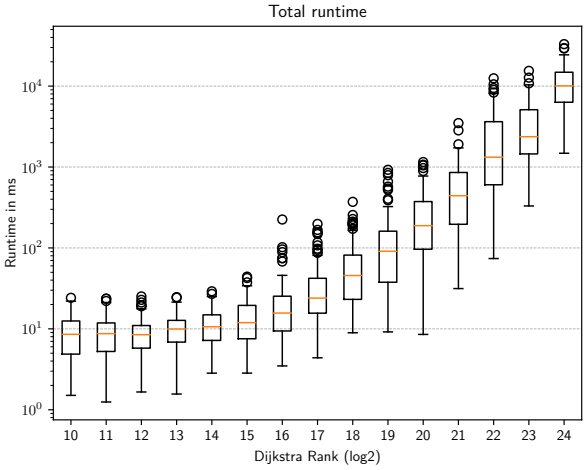


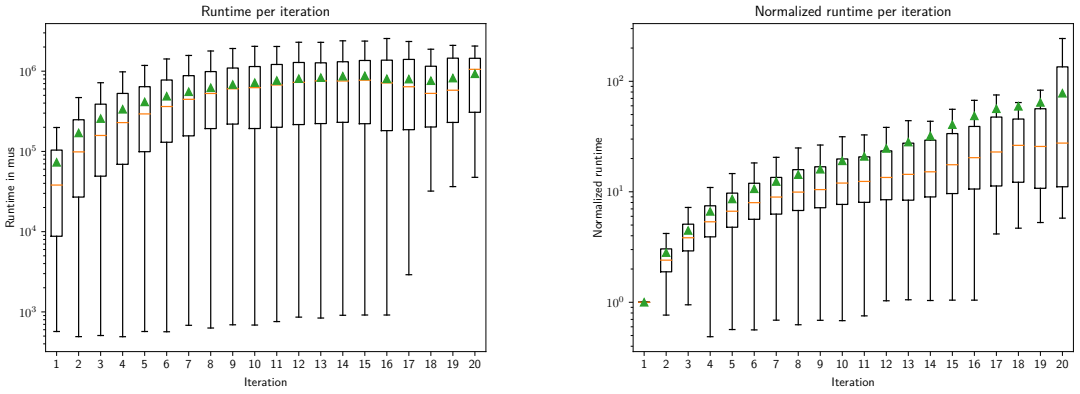
Abb. 2.2: Laufzeit in Abhängigkeit zum Dijkstra-Rank. Sowohl x-Achse als auch y-Achse sind logarithmisch.

Leider können wir die geringen Iterationszahlen von Kobitzsch et al. nicht reproduzieren. Aus Abbildung 1.1 geht hervor, dass selbst für die Dijkstra-Ranks 2^{23} und 2^{24} im Mittel circa 10 bis 12 Iterationen benötigt wird, bis die Penalty-Methode abbricht. Weiterhin schwankt die Anzahl der Iterationen stark.
Durch die hohe Anzahl der Iterationen ist auch die Laufzeit der Penalty-Methode mit A* sehr hoch. In Abbildung 1.2 erkennt man, dass die Laufzeit der Penalty-Methode linear zum Dijkstra-Rank wächst (die x-Achse ist logarithmisch). Mit einer durchschnittlichen Laufzeit von zehn Sekunden und einer Worst-Case-Laufzeit von über 30 Sekunden ist unsere Penalty-Methode etwa um einen Faktor 50 langsamer als die in [KRS13] vorgestellte. Die Qualitätswerte der gefundenen Pfade ist mit denen von Kobitzsch et al. vergleichbar, allerdings ist die Erfolgsquote etwas geringer.
Bei der theoretischen Betrachtung der Ergebnisse aus [KRS13] kommen einige Fragen auf. Die Anzahl der Iterationen der Penalty-Methode ist minimal, wenn nur der kürzeste *st*-Pfad ein sinnvoller *st*-Pfad mit Länge kleiner als $(1 + 0.1) \cdot \mathcal{D}(s, t)$ ist. In diesem Fall würde der *st*-Pfad mehrfach hintereinander mit Pfad-Penaltyfaktor $pen_f = 1.04$ werden. Danach würde ein kürzester Pfad gefunden werden, welcher Länger ist als 10% der Länge des kürzesten *st*-Pfades und die Penalty-Methode bricht ab. Da $1.04^2 < 1.1$ und $1.04^3 > 1.1$ muss der kürzeste st-Pfad mindestens dreimal bestraft werden, bevor die Penalty-Methode abbricht. Die Mindestanzahl an Iterationen ist also drei. Kobitzsch et al. berichten aber von Testfällen mit nur zwei Iterationen bis zum Abbruch.
Wenn es jetzt mehrere gute, weitesgehend vom kürzesten Pfad disjunkte, Alternativrouten gibt, erwarten wir mehr Iterationen, da in jeder Iteration sowohl der original kürzeste Pfad als auch einer der Alternativpfade gefunden und bestraft werden könnte. Je nach Start-Ziel-Knotenpaar gibt es keine solcher Alternativpfade oder einen ganzen Haufen davon. Daher würden wir eine starke Variation der Iterationszahlen, wie wir sie in Abb. 1.1 verzeichnen, erwarten. Gleichzeitig können wir uns die Iterationszahlen von Kobitzsch et al. somit nicht erklären, da die aus dem geringen Iterationszahlen auch eine eher geringe Erfolgsquote folgen müsste, was nicht der Fall ist. Die hohen Iterationszahlen sind für die Laufzeit doppelt fatal: Einerseits muss wegen den hohen Iterationszahlen die Kürzeste-Wege-Anfrage öfters ausgeführt werden. Andererseits wird in jeder Iteration die Heuristik, die durch CH-Potentiale berechnet wird, ungenauer. Somit lassen sich die langsamen Laufzeiten gut durch die Anzahl der Iterationen erklären.

Kriterium	Alternativpfad 1	Alternativpfad 2	Alternativpfad 3
Erfolgsquote	79.1%	61.6%	48.7 %
UBS	8.7%	7.5%	7.5%
Sharing	42.9%	47.5%	51.1%
Local Optimality	26.4%	23.7%	20.4%

Abb. 2.3: Qualitätswerte der gefundenen Alternativpfade. Die Erfolgsquote gibt an, wie häufig ein k-ter Alternativpfad in einer Testinstanz gefunden wurde. Für gefundene Pfade ist UBS, Sharing und Local Optimality wie in Kapitel 2 definiert.

Die Qualitätsparameter der gefundenen Pfade sind ähnlich derer in [KRS13]. Nur die Erfolgsquote ist etwas geringer. Da wir keinen Zugriff auf die Testinstanzen von Kobitzsch et al. haben, ist eine leichte Variaton der Werte nicht zu vermeiden. Auffällig sind allerdings die sehr geringen Werte für UBS. In [KRS13] werden UBS-Werte von durchschnittlich 42.8% für die Penalty-Methode angegeben. Auch diesen Unterschied können wir uns nicht erklären. Kobitzsch et al. erklären ihre hohen UBS-Werte mit dem Auslassen der Tests auf UBS und LO bei der Pfadextraktion mit BDV. Dennoch würden wir zumindest im Durchschnitt geringere Werte für den UBS erwarten, da die Pfade nach Qualität sortiert sind, der Wert für LO von um die 25% schlechte sehr kurze Umwege ausschließt, und an die Pfade, die dem Alternativgraphen hinzugefügt werden, selbst schon gewisse Qualitätsanforderungen gestellt werden.



(a) Laufzeit pro Iteration

(b) Laufzeit pro Iteration, normalisiert

Abb. 2.4: Laufzeit pro Iteration in Mikrosekunden, in (b) normalisiert auf die Laufzeit der ersten Iteration. Die y-Achse ist logarithmisch. Durchschnittswerte sind mit einem grünen Dreieck markiert.

In Abbildung 2.4 kann man erkennen, wie stark die Laufzeit einer einzelnen Iteration pro Iteration wächst. Zusätzlich zum Median sind die arithmetischen Mittel der Werte mit einem grünen Dreieck markiert. Da die Daten aus zufälligen Testinstanzen kommen, schwanken die Laufzeiten in Abb. 2.4 (a) stark. Daher haben wir in Abb. 2.4 (b) die Laufzeiten normalisiert, indem wir sie durch die Laufzeit der ersten Iteration geteilt haben. Wie man sehen kann, wächst die Laufzeit zu Anfang mit jeder Iteration stark an, das Wachstum verringert sich aber mit jeder Iteration. Es ist zu beachten, dass selten mehr als zehn Iterationen erreicht werden, und diese Fälle meistens Anfragen mit geringem Dijkstra-Rank zuzuschreiben sind. Durch die geringe Distanz fallen konstante Störungsfaktoren wie der Overhead für das Multithreading des bidirektionalen Dijkstras stärker ins Gewicht.

Auch ist zu beachten, dass schon vor der ersten Iteration der Graph einmal bestraft wurde. Die erste A*-Suche auf der noch perfekten Heuristik benötigt nur ein paar Millisekunden, wobei der konstante Overhead des Threadings einen Großteil dieser Zeit ausmacht. Daher ist eine Normalisierung auf die erste A*-Suchzeit nicht sinnvoll.

4.3 Realisierung 2: Modifizierte Abbruchbedingung

In dieser Realisierung bricht die Penalty-Methode ab, wenn die Länge des gefundenen kürzesten Pfades auf dem *bestraften* Graphen 10% länger ist als der ursprünglich kürzeste st-Pfad. Zur Erinnerung: In der originalen Abbruchbedingung wird die Pfadlänge auf dem Originalgraphen betrachtet.
Durch eine solche Anpassung des Abbruchkriteriums erwarten wir weitaus weniger Iterationen. In den Logs für die originale Abbruchbedingung bebachteten wir, dass einige Pfade (vor allem der original kürzeste Pfad) sehr häufig besucht wurden. In dieser Abbruchbedingung bricht das Verfahren immer ab, wenn dreimal der selbe Pfad von der Penalty-Methode gefunden wird, da $1.04^3 > 1.1$. Aufgrund der geringeren Iterationszahlen erwarten wir allerdings eine geringere Erfolgsquote zum Finden von Alternativrouten.

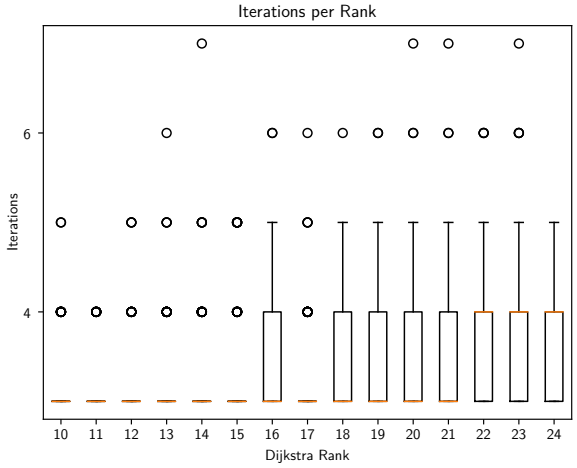


Abb. 3.1: Anzahl der Iterationen in Abhängigkeit zum Dijkstra-Rank. Die x-Achse ist logarithmisch.

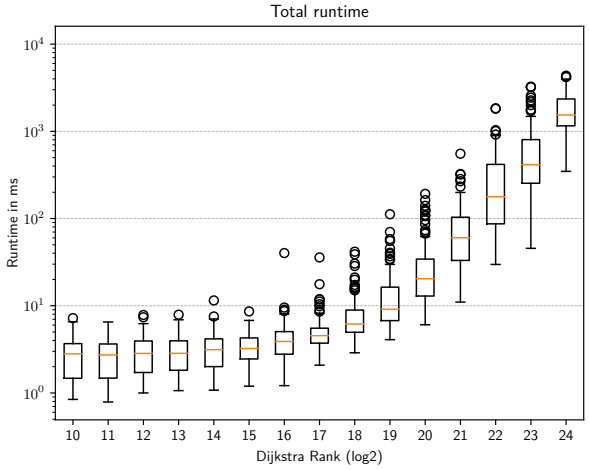


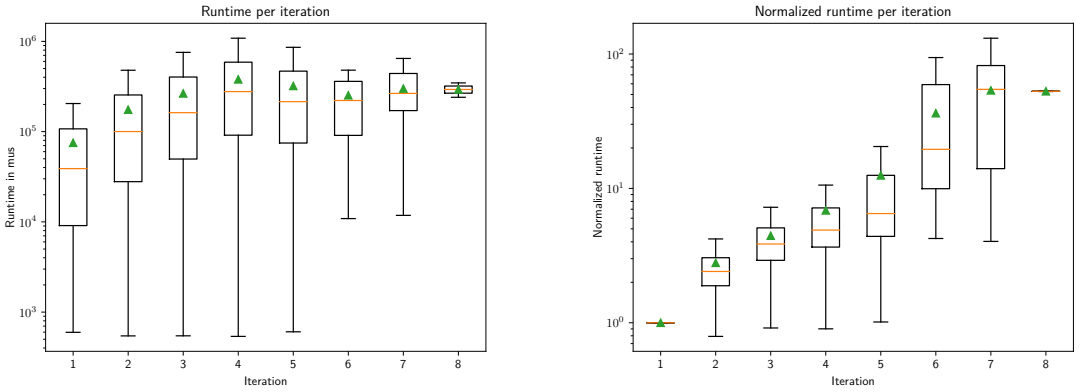
Abb. 3.2: Laufzeit in Abhängigkeit zum Dijkstra-Rank. Sowohl x-Achse als auch y-Achse sind logarithmisch.

Durch die Einführung der alternativen Abbruchbedingung erhalten wir für hohe Dijkstra-Ranks vergleichbare Iterationszahlen wie Kobitzsch et al. Allerdings decken sich die Ergebnisse immer noch nicht mit denen in [KRS13], da nun auch für niedrige Dijkstra-Ranks ($\leq 2^{13}$) nur drei bis vier Iterationen ausgeführt werden, während im Paper für diese Testinstanzen mehr als zehn Iterationen dokumentiert werden.
Durch die geringe Anzahl an Iterationen wurde die Laufzeit der Penalty-Methode um etwa einen Faktor 5 beschleunigt. Für alle außer die größten Dijkstra-Ranks (2^{24}) erreichen wir nun Laufzeiten, welche mit denen in [KRS13] vergleichbar sind. Durch das recht starke Wachstum der Laufzeit in Abhängigkeit zur Pfadlänge ist die Laufzeit für Dijkstra-Rank 2^{24} mit c.a. 1.5 Sekunden im Mittel noch recht langsam. Wie auch in dem originalen Abbruchkriterium schwankt die Laufzeit je nach Verfügbarkeit von Alternativrouten stark.

Kriterium	Alternativpfad 1	Alternativpfad 2	Alternativpfad 3
Erfolgsquote	73.9%	47.1%	27.4 %
UBS	5.4%	4.5%	3.9%
Sharing	45.4%	52.8%	58.3%
Local Optimality	33.0%	27.2%	23.2%

Abb. 3.3: Qualitätswerte der gefundenen Alternativpfade. Die Erfolgsquote gibt an, wie häufig ein k-ter Alternativpfad in einer Testinstanz gefunden wurde. Für gefundene Pfade ist UBS, Sharing und Local Optimality wie in Kapitel 2 definiert.

Überraschenderweise beeinflusst das neue Abbruchkriterium die Qualität der gefundenen Pfade wenig. Die Erfolgsquote, um einen Alternativpfad zu finden ist gerade mal um 4% gesunken. Die Erfolgsquoten für den zweiten und dritten Alternativpfad sind dagegen merklich niedriger als beim originalen Abbruchkriterium. Der erste Alternativpfad weist sehr ähnliche Werte für UBS, Sharing und Local Optimality im Vergleich zum ersten Alternativpfad beim originalen Abbruchkriterium auf. Für die Alternativpfade 2 und 3 sind die Sharing-Werte etwas höher, die Local Optimality-Werte allerdings immer noch vergleichbar mit denen aus dem originalen Abbruchkriterium.



(a) Laufzeit pro Iteration

(b) Laufzeit pro Iteration, normalisiert

Abb. 3.4: Laufzeit pro Iteration in Mikrosekunden, in (b) normalisiert auf die Laufzeit der ersten Iteration. Die y-Achse ist logarithmisch.

Die geringe Anzahl der Iterationen ist natürlich auch in Abbildung 3.4 zu sehen. Die Laufzeiten pro Iteration stimmen mit denen aus Abbildung 2.4 bis zur c.a. fünften Iteration ungefähr überein, danach steigt das Laufzeitwachstum stark an. Das liegt daran, dass viel weniger Testinstanzen mehr als fünf Iterationen benötigen. Die Testinstanzen, die mehr als fünf Iterationen bis zum Abbruch benötigen, beinhalten viele gute Alternativpfade, welche die Laufzeit also stark erhöhen. Im Vergleich: Für die erste Realisierung dagegen sind Iterationszahlen von sechs und mehr Iterationen auch möglich, wenn es recht wenige Alternativpfade gibt.
Diese Realisierung scheint für den praktischen Gebrauch besser geeignet zu sein. Für alle Anfragen außer die längsten Strecken durch Europa erreichen wir interaktive Laufzeiten. Es sollte gesagt sein, dass ein Dijkstra-Rank von 2^{23} etwa einer Strecke durch ganz Deutschland entsprechen könnte (z.B. Flensburg → München). Ein Dijkstra-Rank von 2^{24} impliziert eine der längsten kürzesten Routen durch Westeuropa, z.B. Prag → Stockholm.

4.4 Realisierung 3: Bestrafung des gesamten Alternativgraphen

In den Logs der letzten Realisierung beobachteten wir, dass in den Iterationen häufig wieder zurück auf den Originalpfad gesprungen wird. Das liegt daran, dass der Originalpfad nur um 4% verlängert wird, und es nicht viele Alternativpfade gibt, die nur 4% länger als der Originalpfad sind. Um diese Rücksprungiterationen zu vermeiden, evaluieren wir eine Bestrafungsstrategie, bei welcher der gesamte Alternativgraph in jeder Iteration bestraft wird. In der Penalty-Methode findet daher die Bestrafung *nach* dem Hinzufügen eines Pfades zum Alternativgraphen statt. Die Abbruchbedingung ist wie in Realisierung 1 gewählt.

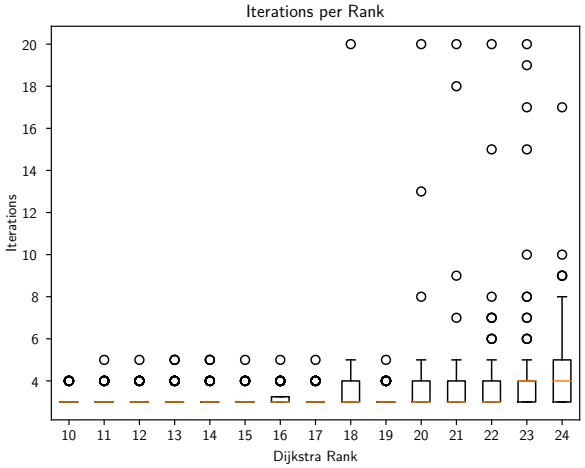


Abb. 4.1: Anzahl der Iterationen in Abhängigkeit zum Dijkstra-Rank. Die x-Achse ist logarithmisch.
(Beachte: Für einige Dijkstra-Ranks ist sowohl das 25te Perzentil als auch das 75te Perzentil drei. Daher hat die Box der Boxplots keine Höhe.)

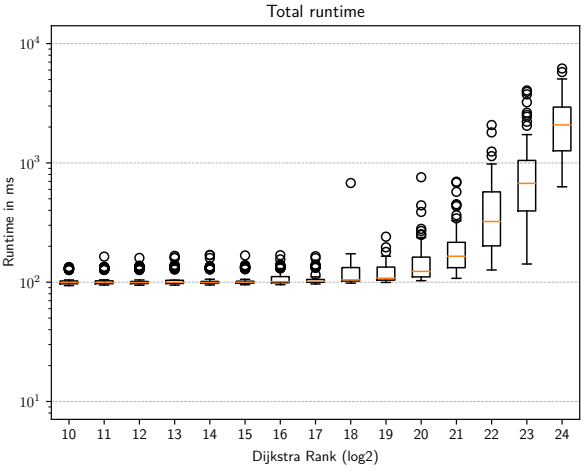


Abb. 4.2: Laufzeit in Abhängigkeit zum Dijkstra-Rank. Sowohl x-Achse als auch y-Achse sind logarithmisch.

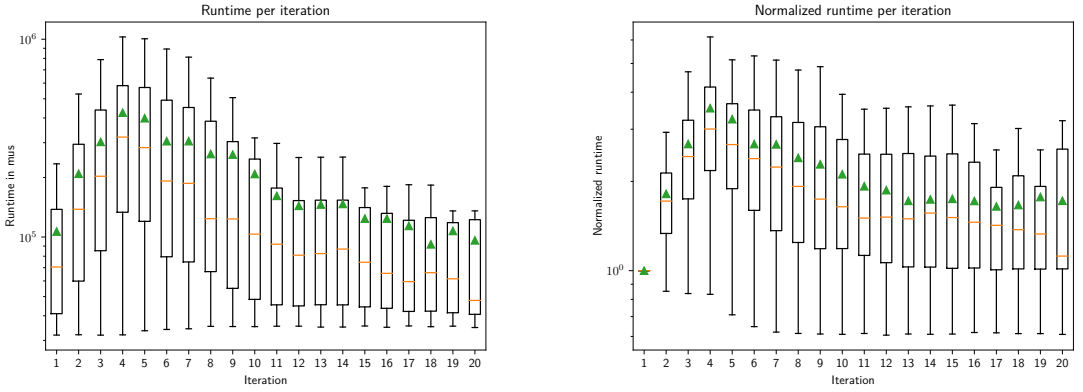
In dieser Realisierung benötigt nun fast jede Testinstanz genau drei oder vier Iterationen. Die Ergebnisse machen nach theoretischer Betrachtung durchaus Sinn: Das Verfahren braucht mindestens drei Iterationen zum Abbrechen, da $1.04^3 \geq 1.1$ und $1.04^2 < 1.1$. Unter der Annahme, dass es nur einen weiteren Alternativpfad gibt, müsste für diesen gelten, dass dessen Länge auch erst nach drei Bestrafungsiterationen das 1.1-fache von $\mathcal{D}_w(s,t)$ übersteigt. Der Alternativpfad darf also höchstens $\frac{1.1}{1.04^2} = 1.017 \implies 1.7\%$ länger sein als $\mathcal{D}_w(s,t)$. Falls der Alternativpfad nicht vom kürzesten Pfad disjunkt ist, muss die Abweichung noch geringer sein, da Teile des Alternativpfades schon in der ersten Iteration bestraft wurden. In der Praxis kommen solch kurze gute Alternativpfade sehr selten vor. Daher sind Iterationszahlen von über vier eher selten. Allerdings gibt es einige Ausreißer, sodass zum Teil bis zu 20 Iterationen benötigt werden. Dies geschieht, da diese Bestrafungsmethode keine Rejoin-Penalty beinhält, sodass viele kleine Abweichungen von der Originalroute in jeder Iteration möglich sind.
Die Laufzeit profitiert von den geringeren Iterationen kaum: Sie wächst immer noch linear

zur Pfadlänge, und durch die Bestrafung des gesamten Graphen wird die Heuristik der CH-Potentiale schneller ungenau als bei Realisierung 2.

Kriterium	Alternativpfad 1	Alternativpfad 2	Alternativpfad 3
Erfolgsquote	52.7%	33.1%	22.2 %
UBS	5.8%	5.0%	5.5%
Sharing	42.9%	47.7%	48.5%
Local Optimality	33.2%	21.2%	17.6%

Abb. 4.3: Qualitätswerte der gefundenen Alternativpfade. Die Erfolgsquote gibt an, wie häufig ein k-ter Alternativpfad in einer Testinstanz gefunden wurde. Für gefundene Pfade ist UBS, Sharing und Local Optimality wie in Kapitel 2 definiert.

Durch die geringen Iterationszahlen werden nur wenige Alternativpfade gefunden. Gerade mal in 52.2% aller Testfälle wurde überhaupt ein Alternativpfad gefunden. Die Erfolgsquoten für weitere Alternativpfade ist entsprechend dürftig. Allerdings ist die Qualität der Pfade, die tatsächlich gefunden werden, immer noch vergleichbar mit denen aus den vorherigen Realisierungen.



(a) Laufzeit pro Iteration

(b) Laufzeit pro Iteration, normalisiert

Abb. 4.4: Laufzeit pro Iteration in Mikrosekunden, in (b) normalisiert auf die Laufzeit der ersten Iteration. Die y-Achse ist logarithmisch.

Da mehr als vier Iterationen eher die Ausnahme als die Regel ist, ist Abbildung 4.4 für die Iterationen fünf und mehr wenig Aussagekräftig. Für die vierte Iteration erkennen wir etwa eine Verdreifachung der Laufzeit im Vergleich zur ersten Iteration. Das ist beachtlich weniger als in den vorherigen Realisierungen, lässt sich aber durch das Fehlen der Rejoin-Penalty, welche Kanten um weitaus mehr als nur 4% bestraft, erklären. Es zeigt sich dadurch allerdings, dass die Rejoin-Penalty einen sehr großen Einfluss auf die Laufzeit der Penalty-Methode mit A* hat.

4.5 Evaluation mit verschiedenen Parametern

Wir evaluieren das Verhalten unserer zweiten Realisierung der Penalty-Methode bei Änderung der Werte pen_f, α und ϵ. pen_f ist der Penalty-Faktor, mit dem die Pfadkanten in jeder Iteration bestraft werden. α ist ein Skalierungsfaktor für die Rejoin-Penalty

$pen_r = \alpha \cdot \sqrt{\mathcal{D}_w(s,t)}$. ϵ gibt an, ab welcher Pfadlänge die Penalty-Methode abbricht. Weiterhin muss, damit das Pfadauswahlkriterium bestanden wird, ein gefundener Pfad mindestens eine Abweichung vom kürzesten Pfad haben, welche einen maximalen Stretch von $1 + \epsilon$ hat.
Die Standardwerte sind $pen_f = 0.04$, $\alpha = 0.5$ und $\epsilon = 0.1$. Wir erwarten, dass ein höherer Penalty-Faktor zu weniger Iterationen aber geringerer Erfolgsquote führt, und ein höherer Wert für ϵ zu mehr Iterationen und einer höheren Erfolgsquote. Da pen_f und α beide Einfluss auf die Kantenbestrafung ausüben, ist das Zusammenspiel der beiden Parameter interessant. Wenn die Rejoin-Penalty zu gering gewählt ist, können Alternativpfade mit vielen kleinen Abweichungen gefunden werden, welche nicht wünschenswert sind und die Anzahl der Iterationen hochtreiben. Ist die Rejoin-Penalty dagegen sehr hoch, kann es sein dass keine neuen Pfade gefunden werden, da es zu teuer wäre, den schon vorhandenen kürzesten Pfad zu verlassen.
Wir evaluieren die Parameterkombinationen $(pen_f, \alpha, \epsilon) \in \{0.03, 0.04, 0.05\} \times \{0.4, 0.5, 0.6\} \times \{0.1\}$ sowie $(pen_f, \alpha, \epsilon) \in \{0.04\} \times \{0.5\} \times \{0.05, 0.1, 0.2\}$ auf Iterationsanzahl, Erfolsquote, Pfadqualität und Laufzeit. Die Qualitätswerte sind jeweils immer für den ersten Alternativpfad, für die Erfolgsquoten geben wir an, wie häufig ein, zwei und drei Pfade gefunden wurden (P1, P2 und P3). Für die Experimente werden aus Zeitgründen nur 100 zufällige Testinstanzen auf unserem Europastraßengraphen getestet.

pen_f	α	Erfolsquote P1	P2	P3	UBS	Sharing	LO	Iter	Laufzeit[ms]
0.03	0.4	66.0%	50.0%	24.0%	5.1%	49.7%	27.5%	4.8	1039
0.04	0.4	66.0%	47.0%	26.0%	5.0%	50.4%	29.7%	3.7	861
0.05	0.4	66.0%	42.0%	22.0%	5.1%	51.1%	30.0%	3.2	794
0.03	0.5	66.0%	49.0%	24.0%	5.2%	49.2%	29.0%	4.7	977
0.04	0.5	65.0%	47.0%	28.0%	5.0%	49.5%	30.3%	3.7	855
0.05	0.5	66.0%	43.0%	21.0%	5.0%	51.0%	30.4%	3.2	802
0.03	0.6	66.0%	49.0%	23.0%	5.2%	49.5%	28.3%	4.7	971
0.04	0.6	65.0%	48.0%	28.0%	5.1%	49.1%	30.0%	3.7	831
0.05	0.6	66.0%	45.0%	23.0%	5.1%	50.8%	30.4%	3.2	816

(a) Variation von pen_f und α, $\epsilon = 0.1$ ist fix

ϵ	Erfolsquote P1	P2	P3	UBS	Sharing	LO	Iter	Laufzeit[ms]
0.05	46.0%	17.0%	3.0%	2.6%	53.3%	37.2%	2.09	299
0.1	65.0%	47.0%	28.0%	5.0%	49.5%	30.3%	3.7	855
0.2	89.0%	76.0%	65.0%	11.2%	38.2%	25.0%	7.8	3182

(b) Variation von ϵ, $pen_f = 1.04$ und $\alpha = 0.5$ sind fix

Abb. 5.1: Durchschnittliche Qualitätswerte für ersten Alternativpfad bei Variation von Parametern der Penalty-Methode

Die Variation von ϵ erfüllt unsere Erwartungen: Bei Halbierung von ϵ halbiert sich die Anzahl der Iterationen und der Wert für den UBS. Die Erfolgsquote verringert sich stark, allerdings wird sie nicht ganz halbiert. Der Sharing-Wert verändert sich nicht maßgeblich mit der Variation von ϵ. Die Local Optimality wird geringer, je höher ϵ ist, da Pfade mit geringem UBS meist auch eine gute LO haben. Die Laufzeit dagegen scheint überproportional zu ϵ zu wachsen. Dies war zu erwarten, denn durch die erhöhte Anzahl von Iterationen wird die Laufzeit gleich dreimal verschlechtert:

- Mehr Iterationen bedeutet mehr Kürzeste-Wege-Queries
- Heuristik wird in jeder Iteration schlechter
- Zu findende Pfade werden in jeder Iteration länger.

Interessanter ist die Variation von pen_f und α. Selbst mit kleinen pen_f und α erhöht sich die Erfolgsquote für den ersten Pfad nicht maßgeblich. Allerdings sind die Erfolgsquoten für Pfad 2 und 3 wie zu erwarten höher, je kleiner ϵ ist. Der Sharing-Wert der gefundenen Pfade scheint sich mit dem Penalty-Faktor leicht zu erhöhen, α dagegen hat wenig bis keinen Einfluss auf alle Qualitätswerte. Wie erwartet ist die Anzahl der Iterationen und damit auch die Laufzeit beachtlich höher, je kleiner pen_f wird. Ein erhöhter α-Wert führt zu einer leichen Beschleunigung des Verfahrens.
Die Werte aus Tabelle 5.1 sind kritisch zu sehen. Es wurden Aufgrund der Vielzahl von Testfällen nur 100 zufällige Knotenpaare ausgewertet, weshalb auch die Erfolgsquoten keine Nachkommastellen aufweisen. Bei einer so geringen Anzahl an Testinstanzen kann der Zufall die Werte maßgeblich beeinflussen, z.B. durch viele Knotenpaare, die überdurchschnittlich nah beieinander liegen. Allerdings wurde jede Instanz auf den selben 100 Knotenpaaren ausgeführt, sodass zwar nicht die Genauigkeit, aber zumindestens die Vergleichbarkeit der Werte in Tabelle 5.1 gewährleistet werden kann.
Zusätzlich wurden nur die Werte für den ersten Alternativpfad ausgewertet. Die Erfolgsquote, ob überhaupt ein Pfad gefunden wird, ist viel stärker von dem (zufällig gewähltem) Knotenpaar abhängig als von den Parametern der Penalty-Methode. In abgeschwächter Form gilt das auch für die Erfolgsquote für die weiteren Pfade. Das erklärt die sehr ähnlichen Erfolgsquoten für den ersten Pfad in jeder Testinstanz.

4.6 Vergleich mit anderen Verfahren

Wir vergleichen unsere Implementation der Penalty-Methode mit verschiedenen Verfahren zur Berechnung von Alternativrouten. Wir nutzen dabei die Verfahren und Werte aus [KRS13] sowie aus [ADGW10]. Die verschiedenen Verfahren werden hinsichtlich Erfolgsquote, Pfadqualität und Laufzeit verglichen. Um die Tabelle übersichtlich zu halten, werden nur die Erfolgsquote und die Qualitätswerte des ersten Alternativpfades gezeigt.

Algo	Erfolgsquote	UBS	Sharing	LO	Laufzeit[ms]
CRP-π-Via	95.2%	42.8%	31.6%	27.1%	N/A ($\approx$150)
X-BDV	94.5%	9.4%	47.2%	73.1%	26352.0
X-REV	91.3%	9.9%	46.9%	71.8%	20.4
X-CHV	58.2%	10.8%	42.9%	72.3%	3.1
Realisierung 1	79.1%	6.1%	46.4%	37.4%	4776.2
Realisierung 2	73.9%	5.4%	45.4%	33.0%	819.2
Realisierung 3	52.7%	5.8%	42.9%	33.2%	1058.6

Abb. 6.1: Vergleich von Realisierung 2 mit anderen Verfahren. Für die Erfolgsquote sowie die Qualitätswerte UBS, Sharing und LO werden die durchschnittlichen Werte des ersten Alternativpfad genommen, für die Laufzeit die durchschnittliche Laufzeit aller zufälligen Testinstanzen.

Wir können erkennen, dass unsere Werte von den Werten von CRP-π-Via ([KRS13]) abweichen. Beachtlich ist auch, dass das Verfahren von Kobitzsch et al. eine höhere Erfolgsquote verspricht als X-BDV, obwohl schlussendlich X-BDV auf den errechneten Alternativgraphen ausgeführt wird. Das lässt sich dadurch erklären, dass die Tests auf Local Optimality und UBS in der Penaltymethode wegfallen, da der Alternativgraph für eine Aussage über diese Parameter zu dünn besetzt ist. Daher kommen auch die sehr hohen Werte für den Uniformly Bounded Stretch.
Im Laufzeitvergleich hinkt unser Verfahren hinterher: Bis auf X-BDV, welches für den direkten praktischen Gebrauch als untauglich angesehen wird, ist jedes andere Verfahren um

mindestens eine Größenordnung schneller als unsere Ergebnisse. Allerdings ist die Qualität der von uns produzierten Pfade vergleichbar oder sogar besser als die der anderen Verfahren. Beachtlich ist, dass wir einen viel geringeren durchschnittlichen Uniformly Bounded Stretch verzeichnen als die Penalty-Methode von Kobitzsch et al. Diese geringen Werte können wir uns bislang nicht erklären.

5. Fazit

Wir haben die Penalty-Methode aus [KRS13] sowie zwei weitere Varianten mithilfe von bidirektionalem A* und CH-Potentialen neu implementiert und evaluiert. Im Vergleich zu anderen Verfahren für die Berechnung von Alternativrouten ist die Penalty-Methode mit bidirektionalem A*, wie sie von uns vorgestellt wurde, recht langsam. Selbst mit geringen Iterationszahlen wird der Graph stark genug verändert, um den Suchraum des bidirektionalen A* stark wachsen zu lassen. Mithilfe von bidirektionalem A* erreichen wir allerdings Laufzeiten von deutlich unter einer Sekunde pro Iteration. Falls wir die Iterationszahlen aus [KRS13] reproduzieren könnten, würden wir vergleichbare Gesamtlaufzeiten erhalten, denn selbst unter Ausnutzung von CRP, wie von Kobitzsch et al. vorgestellt, benötigt eine Iteration noch etwas mehr als 100 Millisekunden für Dijkstra-Ranks über 2^{23}.
Die Penalty-Methode mit A* hat einen sehr geringen Implementationsaufwand. Die CH-Potentiale-Serviceklasse hat in unserer Implementierung nur etwa 50 Standard Lines of Code, die Implementation des bidirektionalen A* etwa 200. Lediglich die Berechnung einer Contraction Hierarchy kann aufwendig sein, allerdings kann man sich auch fertigen Lösungen wie *RoutingKit*[1] bedienen. Weiterhin ist die Laufzeit für kurze und mittellange Strecken durchaus für den interaktiven Gebrauch geeignet, da wir dort Laufzeiten von deutlich unter einer Sekunde erreichen. Für Anwendungen, bei denen die Laufzeit für die längsten Strecken nicht unbedingt interaktiv sein muss, ist unser Algorithmus also geeignet.
Wir können uns die starken Unterschiede zu den Ergebnissen von [KRS13] hinsichtlich Qualität und Iterationszahlen nicht erklären. Es liegt nahe, dass entweder wir oder Kobitzsch et al. einen Fehler in der Implementierung haben, oder einen Schritt ungenau dokumentiert haben. Zumindest für die Iterationszahlen sollte es kaum mögliche Fehlerquellen geben: Wir haben unsere Kürzeste-Wege-Suche mit bidirektionalem A* per Unit-Tests verifiziert. Da Kantenbestrafung und Abbruchkriterium nur etwa 20 Zeilen Code sind, ist es unwahrscheinlich, dass diese Komponenten fehlerbehaftet sind.
Für Interessierte, welche gerne selber experimentieren möchten, ist der Code für unsere Implementation auf `https://github.com/freak532486/pen-astar` verfügbar. Die Benutzung ist in der README dokumentiert. Über Makros, die bei Kompilierung definiert werden können, können die verschiedenen Realisierungen generiert werden. Der gesamte Code steht unter GPL3-Lizenz.

[1] `https://github.com/RoutingKit/RoutingKit`

Literaturverzeichnis

[ADGW10] Ittai Abraham, Daniel Delling, Andrew V. Goldberg und Renato F. Werneck: *Alternative Routes in Road Networks.* In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, Band 6049 der Reihe *Lecture Notes in Computer Science*, Seiten 23–34. Springer, May 2010. `http://research.microsoft.com/apps/pubs/default.aspx?id=121750`.

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg und Renato F. Werneck: *Alternative Routes in Road Networks.* ACM Journal of Experimental Algorithmics, 18(1):1–17, 2013.

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger und Peter Sanders: *Alternative Route Graphs in Road Networks.* In: *Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, Band 6595 der Reihe *Lecture Notes in Computer Science*, Seiten 21–32. Springer, 2011.

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes und Dorothea Wagner: *Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm.* In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, Band 5038 der Reihe *Lecture Notes in Computer Science*, Seiten 303–318. Springer, June 2008.

[Dij59] Edsger W. Dijkstra: *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik, 1:269–271, 1959.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes und Daniel Delling: *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.* In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, Band 5038 der Reihe *Lecture Notes in Computer Science*, Seiten 319–333. Springer, June 2008.

[GW00] Michael T. Goodrich und Christopher G. Wagner: *A Framework for Drawing Planar Graphs with Curves and Polylines.* Journal of Algorithms, 37(2):399–421, 2000.

[GW05] Andrew V. Goldberg und Renato F. Werneck: *Computing Point-to-Point Shortest Paths from External Memory.* In: *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, Seiten 26–40. SIAM, 2005.

[HNR68] Peter E. Hart, Nils Nilsson und Bertram Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, 4:100–107, 1968.

[KRS13] Moritz Kobitzsch, Marcel Radermacher und Dennis Schieferdecker: *Evolution and Evaluation of the Penalty Method for Alternative Graphs* . In: *Proceedings*

of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13), Seiten 94 – 107, 2013.

[SZ21] Ben Strasser und Tim Zeitz: *A Fast and Tight Heuristic for A* in Road Networks*. In: David Coudert und Emanuele Natale (Herausgeber): *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21)*, Band 190 der Reihe *Leibniz International Proceedings in Informatics*, June 2021. To appear.