Berechnung kürzester Pfade unter Beachtung von Abbiegeverboten

Kirill Müller

Studienarbeit am Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. Dorothea Wagner
Universität Karlsruhe, Fakultät für Informatik

27. Oktober 2005

Inhaltsverzeichnis

Zusammenfassung

Die Bestimmung des kürzesten Wegs in einem Graphen ist der Kernbestandteil eines Routenplanungssystems. Für diese Fragestellung existieren Standard-Algorithmen wie der Algorithmus von DIJKSTRA, die im einfachsten Fall den Weg von einem Startknoten zu einem Zielknoten berechnen. Für Graphen, die aus dem Straßennetz eines ganzen Landes erzeugt werden, kann die Laufzeit der Standard-Algorithmen durch verschiedene Beschleunigungstechniken verkürzt werden. Dafür ist eine Vorverarbeitung und/oder die Einbeziehung ergänzender Information wie Geo-Daten notwendig.

Zusätzlich besteht in realen Straßennetzen oft die Einschränkung, dass bestimmte Kantenfolgen nicht durchlaufen werden dürfen – im einfachsten Fall geht es dabei um Abbiege- oder Wendeverbote. Als Erweiterung kann ein Verbot längerer Kantenfolgen betrachtet werden, beispielsweise wenn direkt nach einem Abbiegevorgang kein Spurwechsel und somit kein anschließendes Abbiegen möglich ist. Leider sind Wegeverbote in den Standard-Algorithmen nicht vorgesehen.

In dieser Arbeit untersuchen wir, inwiefern Wegeverbote bei den optimierten Such-Algorithmen berücksichtigt werden können. Das Ziel dabei ist, die bereits bestehenden Algorithmen weitestgehend unverändert zu lassen. Das kann durch eine Transformation des Ausgangsgraphen mit Wegeverboten in einen „Arbeitsgraphen“ ohne Wegeverbote geschehen, auf den dann die bereits extistierenden Algorithmen angewendet werden können. Optimalerweise erfolgt die Transformation on-line während der Berechnung des kürzesten Wegs. Zum Vergleich untersuchen wir die Erweiterung eines Standard-Algorithmus um Wegeverbote.

Ein großer Teil der Arbeit ist theoretischen Aspekten gewidmet. Des weiteren betrachten wir Implementationsdetails und die Einfachheit der Anpassung an vorhandene Gegebenheiten. Zur Evaluierung der praktischen Verwendbarkeit werden Laufzeitmessungen für reale Straßengraphen ausgewertet.

Ausgangspunkt der Arbeit ist die Dissertation von SCHMID [Sch00].

An dieser Stelle möchte ich mich besonders bei meinem Betreuer Dr. Frank Schulz und bei Frau Prof. Dr. Dorothea Wagner für ihre Unterstützung bei der Entstehung dieser Arbeit bedanken. Des weiteren bedanke ich mich bei der PTV AG für die Bereitstellung der Straßen-Daten und bei Andrea Schumm für deren Aufbereitung.

1 Einführung

Dieser Abschnitt soll eine kurze Problembeschreibung bieten und den weiteren Verlauf der Arbeit beschreiben.

1.1 Motivation

Bei der Wahl der Strecke für eine Autofahrt sucht man für gewöhnlich den „besten“ Weg. Für die Ermittlung der Güte eines Wegs können verschiedene Kriterien in Betracht kommen: Erwartete Reisezeit, Länge der Strecke, Kosten für Mautgebühren oder Fähren, mittlere Verkehrsdichte oder vielleicht die Wahrscheinlichkeit einer Radarkontrolle oder die Schönheit der durchfahrenen Landschaft. Je nach Präferenz des Fahrers können die einzelnen Parameter mit unterschiedlicher Gewichtung in eine reellwertige Kostenfunktion eingehen.

Das Routenplanungs-Problem kann leicht auf die Bestimmung eines kürzesten Wegs in einem gewichteten Graphen $G(V, E, c)$ reduziert werden. Dabei entsprechen die Kreuzungen Knoten aus V, die Straßenabschnitte Kanten aus E, und c ist die oben erwähnte Kostenfunktion. Wenn wir von der Länge eines Wegs sprechen, meinen wir also immer die Summe der Kosten der einzelnen Kanten.

1.2 Kürzeste-Wege-Algorithmen

Algorithmen für die Ermittlung des kürzesten Wegs lassen sich in drei Kategorien einteilen:

SSSP „Single Source Shortest Path“-Algorithmen suchen ausgehend von einem Startknoten den kürzesten Weg zu allen Zielknoten. Zur Lösung dieses Problems wird für Graphen mit nichtnegativer Kostenfunktion meist der Algorithmus von Dijkstra eingesetzt. Bei geeigneter Implementation benötigt er einen Zeitaufwand von $O(|E| + |V| \log |V|)$. Wir werden später genauer auf diesen Algorithmus eingehen.

STSP „Source-Target Shortest Path“-Algorithmen ermitteln den kürzesten Weg zwischen zwei Knoten. Für dieses Problem ist kein Algorithmus bekannt, der (im allgemeinen Fall) eine bessere Laufzeit als der beste Algorithmus für das SSSP-Problem aufweist. Daher werden für dieses Problem dieselben Algorithmen wie für das SSSP-Problem eingesetzt und lediglich durch ein Abbruch-Kriterium erweitert. Für Graphen mit nichtnegativer Kostenfunktion kommt also auch hier der Algorithmus von Dijkstra zum Einsatz.

APSP „All Pairs Shortest Path“-Algorithmen bestimmen für alle Knotenpaare den jeweils kürzesten Weg. Der (verblüffend einfache) Algorithmus von Floyd-Warshall (Tripel-Algorithmus) [CLRS01, Flo62] findet in Graphen mit nichtnegativer Kostenfunktion alle kürzesten Wege in $O(|V|^3)$. Bei der Berechnung wird eine $|V| \times |V|$-Matrix aufgebaut, aus der anschließend der Weg zwischen zwei beliebigen Knoten ohne zusätzlichen Aufwand „abgelesen“ werden kann.

Das APSP lässt sich stets auch durch $|V|$-fache Anwendung des SSSP in $O(|V| \cdot (|E| + |V| \log |V|))$ lösen, was für dünne Graphen besser als die Laufzeit des Tripel-Algorithmus ist.

Der Vorteil von APSP-Algorithmen liegt darin, dass nach einer relativ aufwändigen Vorberechnungsphase alle kürzesten Wege jederzeit abrufbereit sind. Für reale Straßengraphen mit Millionen Knoten ist eine solche Vorberechnung jedoch nicht praktikabel: Sowohl die Laufzeit als auch der Speicherbedarf sind immens. Zudem ist es unwahrscheinlich, dass jeder vorberechnete Weg in einem großen Graphen abgefragt wird. Änderungen im Graphen erfordern beim Algorithmus zumindest eine aufwändige Korrektur der vorberechneten Datenstruktur, wenn nicht sogar eine komplette Neuberechnung. Aus diesen Gründen wird man für Straßengraphen von der Verwengung eines APSP-Algorithmus absehen.

Doch auch die SSSP-Algorithmen benötigen für große Graphen sehr viel Zeit. Da bei der Routenplanung fast immer der billigste Weg zu nur einem Zielknoten gesucht ist, würde hier ein

STSP-Algorithmus genügen. Wird der billigste Weg zu nur einem Zielknoten gesucht, können die (meist aus SSSP-Algorithmen gewonnenen) STSP-Algorithmen um Größenordnungen beschleunigt werden, indem eine einmalige Vorberechnungsphase durchgeführt wird und/oder Geo-Daten einbezogen werden. Da wir untersuchen wollen, ob die Berücksichtigung von Abbiegeverboten auch unter Verwendung von Beschleunigungstechniken effizient möglich ist, werden wir weiter unten einige dieser Techniken kurz skizzieren. Ausgangspunkt ist dabei stets der DIJKSTRA-Algorithmus.

1.3 Abbiege- und Wegeverbote

Die Standard-Algorithmen für STSP, wie auch der Algorithmus von DIJKSTRA, erlauben generell das Betreten jeder Kante ausgehend von einem Knoten – unabhängig davon, wie der bisherige Weg verläuft. Im Straßenverkehr werden dem Autofahrer jedoch aus Sicherheitsgründen Einschränkungen in der Bewegungsfreiheit auferlegt.

Im einfachsten Fall wird das Abbiegen und Wenden eingeschränkt. Auf den Straßengraphen abgebildet heißt das, dass bestimmte Kantenfolgen der Länge 2 nicht durchlaufen werden dürfen – es liegt ein *Abbiegeverbot* vor.

Diese Einschränkung kann man auf verbotene Kantenfolgen beliebiger Länge ausweiten. In [Sch00] wurde zur Motivation ein Beispiel präsentiert, bei dem ein Spurwechsel in die linke Spur nach einem Abbiegemanöver nach rechts nicht möglich ist, so dass ein anschließendes Linksabbiegen nicht in Frage kommt. Das kann als ein *Wegeverbot* der Länge 3 abgebildet werden.

1.4 Aufbau der Arbeit

Im weiteren Verlauf der Arbeit befassen wir uns mit der Fragestellung, auf welche Art und Weise Abbiege- und Wegeverbote bei der Suche des kürzesten Wegs berücksichtigt werden können. In Abschnitt 2 stellen wir den Algorithmus von DIJKSTRA vor, der als Ausgangspunkt für alle von uns betrachteten Kürzeste-Wege-Algorithmen dient. Wir skizzieren einige Möglichkeiten der Beschleunigung und führen einige Begriffe ein. Abschnitt 3 beschäftigt sich mit einer Variante der Kürzesten-Wege-Suche, die wir im weiteren Verlauf der Arbeit benötigen werden: Wir lassen jeweils mehr als einen Start- bzw. Zielknoten zu. In Abschnitt 4 stellen wir vier verschiedene Verfahren vor, die die Kürzeste-Wege-Suche unter Berücksichtigung von Abbiegeverboten realisieren. Dabei werden wir teils den Algorithmus verändern und teils die Abbiegeverbote in den Graphen einbauen, so dass bereits ein Standard-Algorithmus die Abbiegeverbote berücksichtigt. Wir weisen die Korrektheit der Verfahren nach und betrachten die Komplexität. Ferner gehen wir hier kurz auf allgemeine Wegeverbote mit mehr als zwei Kanten ein. Unsere Referenz-Implementation stellen wir in Abschnitt 5 vor. Wir umreißen die dabei aufgetretenen Schwierigkeiten, um auf die bei einer anwendungsorientierten Implementation zu erwartenden Probleme aufmerksam zu machen. Es folgen empirische Auswertungen in Abschnitt 6: Wir vergleichen die verschiedenen Verfahren zur Berücksichtigung von Abbiegeverboten und führen einige Statistiken an. Ein Ausblick in Abschnitt 7 schließt die Arbeit ab. Im Anhang zeigen wir Pseudo-Codes der vorgestellten Algorithmen, ein Ablaufbeispiel und die Grammatik für unser Graph-Dateiformat.

2 Der Algorithmus von DIJKSTRA

In diesem Abschnitt werden wir zunächst den Algorithmus von DIJKSTRA vorstellen, der als Basis für alle in dieser Arbeit vorgestellten Algorithmen dient. Im Anschluss werden wir kurz darauf eingehen, wie man ihn beschleunigen kann. Im Unterabschnitt 2.4 besprechen wir Strategien zur Berücksichtigung der Abbiegeverbote. Doch zunächst möchten wir einige formale Aspekte anführen und die verwendeten Objekte und Operationen definieren.

2.1 Formales

Im Folgenden werden wir die bei einer Suchanfrage verwendeten Objekte wie folgt benennen:

- $G(V, E, c)$: Graph mit Knotenmenge V, Kantenmenge E und Kostenfunktion $c : E \to \mathbb{R}_+$.
- $v_s, v_t \in V$: Start- und Zielknoten.
- $P = \langle e_1, \ldots, e_n \rangle$: *Pfade* sind als (möglicherweise leere) Kantenfolgen definiert.

Zusätzlich definieren wir der Klarheit halber die verwendeten Begriffe und Operationen:

- *Start-* und *Zielknoten* einer Kante: Für $e = (v_0, v_1) \in E$ ist $source(e) = v_0$ und $target(e) = v_1$.
- *Eingangs-* und *Ausgangskantenmenge* eines Knotens: Für $v \in V$ ist $in(v) = \{e \in E : target(e) = v\}$ und $out(v) = \{e \in E : source(e) = v\}$.
- *Eingangs-* und *Ausgangsgrad* eines Knotens: Für $v \in V$ definieren wir $indeg(v) = |in(v)|$ und $outdeg(v) = |out(v)|$.
- *Gültigkeit eines Pfades*: Ein Pfad $P = \langle e_1, \ldots, e_n \rangle$ ist gültig, wenn stets $target(e_i) = source(e_{i+1})$ gilt.
- *Start- und Zielknoten eines Pfades*: Ein Pfad $P = \langle e_1, \ldots, e_n \rangle$ hat $source(e_1)$ als Start- und $target(e_n)$ als Zielknoten.
- *Kosten eines Pfades*: Für einen Pfad $P = \langle e_1, \ldots e_n \rangle$ ergeben sich seine Kosten $c(P)$ aus der Summe der Einzelkosten $\sum_{i=1}^{n} c(e_i)$.

Wir untersuchen in dieser Arbeit vorwiegend Algorithmen, die auf Straßengraphen zum Einsatz kommen sollen. Wir nehmen daher stets $|E| = O(|V|)$ an und vereinfachen unsere Laufzeitabschätzungen entsprechend, wobei wir die Laufzeit für dünne Graphen mit dem Gleichheits-Operator $\cong$ angeben und $|V|$ durch n ersetzen. (Für einige Länder Europas können wir die Anzahl Kanten relativ genau eingrenzen: $2{,}1 \cdot |V| < |E| < 2{,}4 \cdot |V|$, siehe auch Abschnitt 6.1.)

Für unsere Graphen fordern wir, dass sie keine Multikanten enthalten, also dass zwischen je zwei Knoten höchstens eine Kante definiert ist. Durch das Einfügen geeigneter zusätzlicher Knoten kann ein Graph mit Multikanten stets in einen äquivalenten Graphen ohne Multikanten transformiert werden.

Ferner gehen wir im Folgenden stillschweigend davon aus, dass unsere Graphen stark zusammenhängend sind, dass es also für alle Knotenpaare mindestens einen Weg vom einen zum anderen Knoten gibt. (Insbesondere hat jeder Knoten mindestens eine eingehende und mindestens eine ausgehende Kante.) Die Prüfung des starken Zusammenhangs kann mittels einer zweifach durchgeführten Tiefensuche mit einem einmaligen Laufzeit- und Speicher-Aufwand von $O(|V| + |E|) \cong O(n)$ geschehen [CLRS01].

Außerdem fordern wir, dass die Kostenfunktion c für alle Kanten nichtnegativ ist. Ein Hinweis zum Umgang mit negativen Kostenfunktionen ist in [WW05] angegeben: Der Algorithmus von Johnson [Joh77] berechnet zu einer gegebenen Kostenfunktion mit negativen Werten eine äquivalente nichtnegative Kostenfunktion, so dass die kürzesten Wege unverändert bleiben.

Wir setzen voraus, dass in jeder Knotenmenge V ein dediziertes Element nil_V existiert und eindeutig bestimmt ist. Dieses Element ist mit keiner Kante (außer nil_E) verbunden und dient unseren Algorithmen als Dummy-Element. Entsprechend fordern wir für alle Kantenmengen E die Existenz eines Elements $nil_E := (nil_V, nil_V)$. Den Index lassen wir meist weg und schreiben einfach *nil*. Die Dummy-Elemente sind von unserer Forderung bezüglich des starken Zusammenhangs ausgenommen.

2.2 Überblick

2.2.1 Beschreibung

Der Algorithmus von Dijkstra wird in vielen Standard-Werken über Algorithmen detailliert vorgestellt [CLRS01, Sed88, AHU83], wir geben auch einen Verweis auf die Original-Arbeit an [Dij59]. Er soll hier nur kurz beschrieben werden.

Ausgehend von einem Startknoten v_s werden die kürzesten Wege zu allen Zielknoten berechnet. Kernbestandteil des Algorithmus ist eine Prioritätswarteschlange Q, die Knoten aufsteigend sortiert nach dem bisher ermittelten kürzesten Abstand zum Ziel aufnimmt.

Die Prioritätswarteschlange Q wird mit $(v_s, 0)$ initialisiert. Anschließend wird in jedem Schritt der Knoten v_{min} mit dem kürzesten Wert für den Abstand zum Ziel aus Q extrahiert, und Q wird durch Hinzufügen neuer Knoten oder durch Aktualisieren der Priorität bestehender Knoten aktualisiert.

Die neu hinzugefügten Knoten werden aus den direkten Nachbarn von v_{min} ausgewählt. Ein Knoten v_{neu} wird genau dann hinzugefügt oder aktualisiert, wenn der Weg über v_{min} kürzer ist als der bisher bekannte kürzeste Weg zu v_{neu}. Die Wegkosten des Wegs über v_{min} berechnen sich (wegen der Additivität der Wegkosten) aus der Summe des Abstands zwischen v_s und v_{min} und der Kantenlänge zwischen v_{min} und v_{neu}.

Wenn bisher kein Weg zu v_{neu} ermittelt wurde, ist v_{neu} auch nicht in der Prioritätswarteschlange und wird einfach hinzugefügt. Andernfalls muss der Knoten in Q gefunden und seine Priorität aktualisiert werden. Dabei wird die Priorität innerhalb der Warteschlange immer derart verändert, dass der Knoten an derselben Position bleibt oder an eine weiter vorn gelegene Position verschoben wird. Wir speichern dabei die Kosten $C_s[v]$ des bisher ermittelten kürzesten Wegs zu v.

Der Algorithmus terminiert bei leerer Prioritätswarteschlange oder (im Fall STSP) bei Erreichen des Zielknotens.

Durch Festhalten der Vorgängerkante $P_s[v]$, über den ein kürzester Weg zu einem Knoten v führt, kann neben den Wegkosten auch der tatsächliche Wegverlauf ermittelt werden. Die Pflege dieser Information ändert nichts an der asymptotischen Laufzeit.

Beim Entfernen eines Knotens v_{min} aus Q gilt folgende Invariante: Für alle Knoten, die eine geringere Entfernung zum Ziel haben als v_{min}, ist der kürzeste Weg bereits berechnet. Daher kann man sich die Arbeitsweise des Algorithmus derart vorstellen, dass um den Startknoten herum ein Suchhorizont aufgebaut und ausgeweitet wird, bis der Zielknoten erreicht wird oder alle erreichbaren Knoten betrachtet wurden.

Wir geben im Anhang A auf Seite 40 den Algorithmus für den Fall STSP als Pseudocode an.

2.2.2 Komplexität

Die Laufzeit beträgt für allgemeine Graphen $O(|E|+|V|\log|V|) \cong O(n \log n)$, wenn für die Prioritätswarteschlange ein Fibonacci-Heap verwendet wird und der Graph als Array von Adjazenzlisten vorliegt. Für eine weitergehende Analyse verweisen wir auf [CLRS01].

Die Vorgänger- und Kostenvektoren benötigen $O(|V|)$ Speicher. Die Prioritätswarteschlange kann ebenfalls maximal $|V|$ Knoten enthalten. Dasselbe gilt für die Kantenanzahl im kürzesten Weg P. Demnach beläuft sich der Speicheraufwand für den Algorithmus von DIJKSTRA auf $O(|V|) \cong O(n)$.

2.2.3 Zugriff auf den Graphen

Für gewöhnlich liegt ein dünner Graph als Array von Adjazenzlisten im Arbeitsspeicher der Anwendung. Das ermöglicht einerseits einen Speicheraufwand von $O(|V|+|E|) \cong O(n)$ und andererseits eine effiziente Ausführung des Algorithmus von DIJKSTRA.

Da bei uns auch Fälle auftreten werden, wo der Graph erst bei der Ausführung des Algorithmus erzeugt wird (siehe Abschnitt 2.4), möchten wir bereits hier untersuchen, auf welche Art und Weise auf den Graphen zugegriffen wird. Bei einer „faulen“ Erzeugung des Graphen müssen wir nämlich mindestens diesen Umfang an Zugriffsmethoden zur Verfügung stellen.

Wir beziehen uns dabei auf den Pseudocode des Algorithmus im Anhang A auf Seite 40. Dort sind die hier angegebenen Zugriffsoperationen unterstrichen. Wenn nicht anders angegeben, streben wir eine Laufzeit von $O(1)$ für jede der folgenden Zugriffsoperationen an:

Abfrage auf Gleichheit zweier Knoten bzw. Kanten Wir gehen davon aus, dass der Test auf Gleichheit für Knoten bzw. Kanten kein Zusatzwissen erfordert und z.B. generisch durch einen einfachen Zeiger-Vergleich realisiert werden kann.

Alle ausgehenden Kanten (Zeile 30): $out(v_{min})$ in Laufzeit $O(|out(v_{min})|)$ ($O(1)$ pro Kante)

Start- bzw. Zielknoten einer Kante (Zeilen 24 bzw. 31): $source(e)$ bzw. $target(e_{neu})$

Kosten einer Kante (Zeile 32): $c(e_{neu})$

Man beachte, dass wir innerhalb der äußeren Schleife beim Algorithmus von DIJKSTRA keine Sicht auf die gesamte Knotenmenge V benötigen. Die Initialisierung von C_s und P_s in Zeile 2 iteriert jedoch über alle Knoten. Bei einer realen Implementation verzichten wir auf diese Iteration: Statt dessen verwenden wir für C_s und P_s eine Wörterbuch-Datenstruktur, die anfangs leer ist. Bei der Abarbeitung des Algorithmus prüfen wir beispielsweise für die Abfrage von $C_s[v]$, ob v im Wörterbuch C_s enthalten ist. Wenn nicht, bedeutet das $C_s[v] = \infty$. Eine Zuweisung eines Werts an $C_s[v]$ kann eine Einfüge- oder eine Aktualisierungs-Operation im Wörterbuch nach sich ziehen. Entsprechend gehen wir für P_s vor. Um die asymptotische Laufzeit nicht zu verändern, empfiehlt sich hier die Benutzung einer Hash-Tabelle mit amortisierter Laufzeit von $O(1)$ für die Operationen Abfrage und Aktualisierung [CLRS01].

Es genügt also, die vier oben angegebenen Operationen zu definieren, damit der Algorithmus von DIJKSTRA angewendet werden kann. Die Beschränkung der Laufzeit dieser Operationen ist so gewählt, dass die Laufzeit des DIJKSTRA-Algorithmus im Vergleich zur Anwendung auf einem einfachen Graphen nicht verändert wird.

2.3 Beschleunigungstechniken

2.3.1 Überblick

In [WW05] werden fünf Verfahren vorgestellt, mit denen die Laufzeit des Algorithmus von DIJKSTRA vor allem auf Straßengraphen reduziert werden kann. Ein weiterer, neuerer Ansatz ist in [SS05] zu finden. Vor kurzem wurde die Kombination zweier in [WW05] vorgestellter Techniken untersucht [GKW05]. Wir werden diese Verfahren kurz skizzieren.

Bigerichtete Suche Die *bigerichtete Suche* (bidirectional search) [Poh69] benutzt nicht nur den Start-, sondern auch den Zielknoten als Ausgangspunkt der Suche. Hier werden zwei Prioritätswarteschlangen benötigt, außerdem ist das Abbruch-Kriterium etwas komplizierter. Dadurch lässt sich die Anzahl der betrachteten Knoten in der Praxis etwa halbieren, wenn Start- und Zielknoten im Graphen nah beieinander liegen. Diese Beschleunigungstechnik benötigt keine weiteren Informationen über den Graph.

Zielgerichtete Suche Die *zielgerichtete Suche* oder A^*-Suche (goal-directed search) [HNR68] fügt jedem Knoten ein Potential hinzu. Das Potential fließt bei der Berechnung der Position des Knotens in der Prioritätswarteschlange mit ein. Dadurch werden Knoten mit einem geringeren Potential eher betrachtet, was zu einer „Faltung“ des Suchraums in Richtung Ziel führt. Das Potential kann aus einem Layout des Graphen in der Ebene oder durch eine vorangehende Vorberechnungsphase gewonnen werden.

Aufteilung in Ebenen Bei der *Aufteilung in Ebenen* (multi-level approach) [Hol03] werden die Knoten des Graphen iterativ auf mehrere Ebenen verteilt. Zusätzlich werden Abkürzungskanten innerhalb der Ebenen eingefügt, die freilich nichts an der Länge der kürzesten Wege ändern. Der kürzeste Weg innerhalb einer Ebene kann jedoch mit Hilfe dieser Kanten berechnet werden, ohne die Ebene zu verlassen. Man unterscheidet jetzt bei den Kanten drei Typen: Aufwärts-, Abwärts- und Ebenen-Kante. Es kann effizient entschieden werden, ob eine Aufwärts- bzw. Abwärts-Kante überhaupt betrachtet werden muss.

Routing basierend auf Reichweite Beim *Routing basierend auf Reichweite* (reach-based routing) [Gut04] werden „zentral gelegene Knoten“ bevorzugt. Ein Knoten wird als zentral angesehen, wenn es lange kürzeste Pfade gibt, bei denen dieser Knoten nah bei der Mitte des Pfades liegt. Bei der Ausführung des Algorithmus von DIJKSTRA können Knoten ignoriert werden, wenn sie nicht „zentral genug“ liegen. Für diese Technik ist eine aufwändige Vorberechnung in Form einer modifizierten APSP-Suche notwendig.

Kantenmarkierungen Die *Kantenmarkierungen* (edge labels) beschleunigen die Suche dadurch, dass in einer Vorberechnungsphase zu jeder Kante die Knoten ermittelt werden, zu denen der kürzeste Weg vom Startknoten der Kante über diese Kante führt. Dabei genügt das Festhalten einer Obermenge, beispielsweise die Angabe des kleinsten umschließenden Rechtecks [WW03] oder bei vorhergehender Partitionierung des Graphen die Angabe der Partitionen als Bitvektor [Sch05b]. Diese Menge nennen wir Zielmenge der Kante. Eine Kante braucht nicht betrachtet zu werden, wenn der Zielknoten nicht in der Zielmenge liegt: In diesem Fall kann der kürzeste Weg sicher nicht über diese Kante führen.

Autobahn-Hierarchien Die in [SS05, Sch05a] vorgestellten *Autobahn-Hierarchien* (highway hierarchies) benötigen ebenfalls eine Vorverarbeitungsphase in Linearzeit. Darin werden mit Hilfe eines einfachen Kriteriums „wichtige“ Kanten ermittelt. Der aus wichtigen Kanten bestehende Teilgraph kann vereinfacht werden. Ähnlich wie bei der Aufteilung in Ebenen kann dieser Prozess rekursiv fortgesetzt werden. Auf der so gewonnenen Graphen-Hierarchie wird eine Modifikation des bidirektionalen Algorithmus von DIJKSTRA angesetzt.

Kombination von zielgerichteter und Reichweiten-Suche In [GKW05] wurde die zielgerichtete Suche (in der Variante mit Vorberechnung des Potentials) mit dem Routing basierend auf Reichweite kombiniert („Reach for A^*“). Der Aufwand der Vorverarbeitung bei der zielgerichteten Suche wurde ebenfalls auf Linearzeit reduziert.

2.3.2 Kombination von Beschleunigungstechniken

In [WW05] wird diskutiert, inwiefern sich Beschleunigungstechniken kombinieren lassen. Offenbar lassen sich die ersten fünf der oben erwähnten Beschleunigungstechniken mit mehr oder weniger Aufwand paarweise kombinieren. Beispiele für Kombinationen von Verfahren sind die Autobahn-Hierarchien (bei denen ein neuer Ansatz mit einem bekannten kombiniert wird) und „Reach for A^*“ (wo zwei bekannte Verfahren kombiniert und verbessert werden). Dabei scheinen die kombinierten Verfahren sehr viel effizienter zu sein als die jeweiligen Beschleunigungstechniken für sich.

2.3.3 Zugriff auf den Graphen

Zusätzlich zu den vom Algorithmus von DIJKSTRA benötigten Zugriffsoperationen müssen bei den verschiedenen Beschleunigungstechniken die folgenden Informationen effizient abgefragt werden können:

Alle eingehenden Kanten Bei der bigerichteten Suche wird der Weg rückwärts ausgehend vom Zielknoten betrachtet. Daher benötigen wir hier zusätzlich eine Berechnung der eingehenden Kanten $in(v)$ zu einem Knoten v in Laufzeit $O(|in(v)|)$.

Knoten- und Kantenattribute Wenn Attribute zu Knoten oder Kanten verwaltet werden, wie das z.B. bei der zielgerichteten Suche der Fall ist, müssen diese Attribute in jeweils $O(1)$ Laufzeit nachgeschlagen werden können.

Konstruktion des Graphen Genügt die eingeschränkte lokale Sicht auf den Graphen nicht, kann mit einer einfachen Tiefensuche der gesamte Graph konstruiert und somit jegliche Form der Vorverarbeitung realisiert werden. Die Tiefensuche benötigt $O(|V| + |E|) \cong O(n)$ Laufzeit und Speicher, was nicht teurer ist als jede tiefer gehende Vorverarbeitung.

2.4 Vorgehensweisen zur Berücksichtigung von Abbiege- und Wegeverboten

Für die Berücksichtigung von Wegeverboten bei der Suche nach dem kürzesten Weg sind verschiedene Vorgehensweisen denkbar:

1. *Dynamisch*: Anpassung des Algorithmus, so dass nur noch Wege betrachtet werden, die keine verbotenen Teilwege enthalten.

2. *Statisch*: Informationserhaltende Umwandlung des Graphen mit Wegeverboten in einen *Arbeitsgraphen* ohne Wegeverbote, auf dem der unveränderte Algorithmus arbeiten kann. Die Umwandlung kann dabei

   - *off-line* durch vorherige Berechnung
   - *on-line* während der Abarbeitung des Algorithmus

   geschehen.

Die Begriffe „dynamisch“ und „statisch“ haben wir aus [Sch00] übernommen. Die Wahl der Begriffe kann damit erkärt werden, dass bei statischen Verfahren die Abbiegeverbote bereits (statisch) in den Graphen eingearbeitet sind und somit nicht mehr betrachtet werden müssen, außerdem bleibt der Kürzeste-Wege-Algorithmus unverändert (also auch statisch). Dynamische Verfahren sind der Gegensatz zu statischen Verfahren, da der Algorithmus modifiziert wird und während der Abarbeitung auf die Abbiegeverbote zugreift.

Der Zeitpunkt der Konstruktion des Graphen wird mit „on-line“ bzw. „off-line“ angegeben. Diese Begriffe werden in einer ähnlichen Bedeutung bei Entscheidungsverfahren verwendet [MR95]: On-line-Verfahren erhalten die Anfragen stückweise und müssen nach jeder Anfrage separat entscheiden, während off-line-Verfahren vor ihrer Entscheidung die gesamte Anfrage zu sehen bekommen. Bei uns heißt off-line, dass das Verfahren durch Vorberechnung des Graphen auf alle möglichen Anfragen „vorbereitet“ wird. On-line bedeutet, dass bei jeder Anfrage „neu entschieden“ werden muss.

Wir werden sowohl dynamische als auch statische Verfahren vorstellen. Der klare Vorteil eines statischen Verfahrens liegt darin, dass der Anpassungsaufwand vorhandener Implementationen von Kürzeste-Wege-Algorithmen sehr gering ist. Der Idealfall wäre ein statisches Verfahren, das auf viele vorhandene Implementationen ohne Anpassungsaufwand aufgesetzt werden kann. Wir werden versuchen, diesem Ideal so nahe wie möglich zu kommen.

Um ein statisches on-line-Verfahren zu realisieren, auf den der Algorithmus von Dijkstra aufgesetzt werden kann, müssen die in Unterabschnitt 2.2.3 aufgelisteten Zugriffsoperationen definiert und effizient ausführbar sein. Für die Beschleunigungstechniken müssen zusätzlich die jeweils benötigten Zugriffsoperationen aus Unterabschnitt 2.3.3 erklärt sein. Zu jedem on-line-Verfahren, das die „Dijkstra-Operationen“ definiert, kann generisch das zugehörige off-line-Verfahren erzeugt werden kann, indem der Arbeitsgraph per Tiefensuche konstruiert wird.

Die on-line-Verfahren gegenüber den off-line-Verfahren den Vorteil, dass bei Änderungen am Graphen oder an den Abbiegeverboten keine vorberechnete redundante Datenstruktur aktualisiert werden muss. Ein Nachteil von on-line-Verfahren ist, dass die Suche länger dauern kann, wenn die „Dijkstra-Operationen“ auf dem Arbeitsgraphen laufzeitintensiv sind. Dabei geht es allerdings nur um konstante Faktoren, da wir die asymptotische Komplexität der Zugriffsoperationen beschränkt haben. Wir werden bei den beiden vorgestellten statischen Verfahren immer gleich die on-line-Variante angeben, um die Wahlfreiheit zwischen on-line und off-line zu bewahren.

3 MSTSP: Mehrere Start- und Zielknoten beim STSP

Wir möchten an dieser Stelle vorwegnehmen, dass das oben erwähnte Idealziel eines statischen Verfahrens nur erreicht werden kann, wenn erhebliche Abstriche bei der Laufzeit und bei der Größe

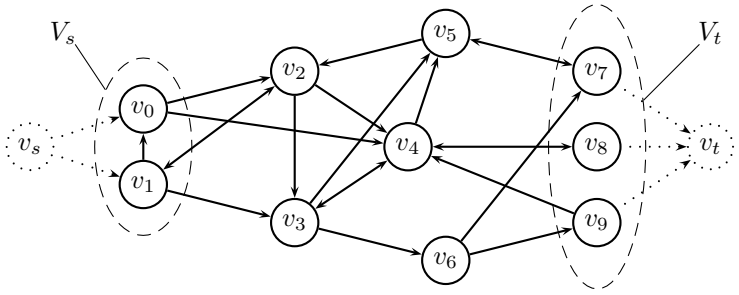


Abbildung 1: Beispiel für die statische Transformation eines Graphen zur Lösung des MSTSP

des verwendeten Arbeitsspeichers hingenommen werden. Ein Problem aller on-line-Verfahren ist, dass Start- und Zielknoten nicht mehr eindeutig festgelegt sind.

Das Problem MSTSP (Mehrfach-STSP) definieren wir wie folgt: Gegeben seien eine Startknotenmenge V_s und eine Zielknotenmenge V_t. Unter allen Wegen, die einen Knoten aus V_s mit einem Knoten aus V_t verbinden, ist der kürzeste gesucht.

Die Lösung dieses Problems kann sowohl durch statische Änderung des Graphen als auch durch dynamische Anpassung des Algorithmus erfolgen. Wir werden beide Ansätze skizzieren und bewerten.

3.1 Allgemeiner statischer Ansatz

3.1.1 Beschreibung

Zu einem Graphen $G(V, E, c)$, einer Startknotenmenge V_s und einer Zielknotenmenge V_t definieren wir einen modifizierten Graphen $G_X(V_X, E_X, c_X)$ wie folgt:

- $V_X := V \uplus \{v_s, v_t\}$: Wir erzeugen zwei Dummy-Knoten, je einen für Start und Ziel.
- $E_X := E \cup \{(v_s, v) : v \in V_s\} \cup \{(v, v_t) : v \in V_t\}$: Für jeden Knoten aus der Startknotenmenge definieren wir eine Kante von v_s zu diesem Knoten, analog symmetrisch für v_t.
- $c_X(e) := \begin{cases} c(e): & e \in E \\ 0: & \text{sonst} \end{cases}$: Die Kosten der neuen Kanten betragen Null, sonst ändert sich nichts.

Abbildung 1 zeigt ein Beispiel. Die in Frage kommenden Start- bzw. Zielknoten sind von einer gestrichelten Linie umgeben. Die bei der Transformation neu hinzugekommenen Knoten und Kanten sind mit gepunkteten Linien dargestellt.

3.1.2 Korrektheit

Für jeden gültigen Pfad $P_X = \langle e_1, \ldots, e_n \rangle$ in G_X von v_s zu v_t gilt dabei $target(e_1) \in V_s$ und $source(e_n) \in V_t$. Man beachte, dass die Knoten v_s und v_t nur als Start- oder Zielknoten im Pfad vorkommen können. Es ist nämlich $in(v_s) = \emptyset$, somit kann es keine Kante e_i mit $target(e) = v_s$ geben. Entsprechendes gilt symmetrisch für v_t.

Somit kann mit $P := \langle e_2, \ldots, e_{n-1} \rangle$ jedem gültigen Pfad in G_X zwischen v_s und v_t ein gültiger Pfad in G zugeordnet werden, der nur Knoten aus V und also nur Kanten aus E enthält. P verbindet einen Knoten aus V_s mit einem Knoten aus V_t. Umgekehrt kann jeder gültige Pfad P zwischen einem Knoten aus V_s und einem Knoten aus V_t zu einem Pfad P_X zwischen v_s und v_t erweitert werden. Alle Pfade zwischen v_s und v_t in G_X erzeugen also genau alle Pfade zwischen allen Knotenpaaren aus $V_s \times V_t$.

Für einen Pfad P_X zwischen v_s und v_t und für den entsprechenden Pfad P gilt $c_X(P_X) = c(P)$. Also erzeugt der kürzeste Pfad zwischen v_s und v_t in G_X den kürzesten Pfad zwischen allen Paaren aus V_s und V_t in G. □

3.1.3 Komplexität

Bei der Transformation kommen zwei neue Knoten und $|V_s|+|V_t|$ neue Kanten hinzu. Für unseren Anwendungsfall bei dünnen Graphen sind Start- und Zielknotenmenge „klein“, also in der Größenordnung von $O(1)$. Demnach lässt sich die Transformation mit konstantem Aufwand durchführen.

Die on-line-Transformation ist ebenfalls möglich. Dazu muss lediglich die Aufzählung der Start- und der Test auf Enthaltensein in der Zielknotenmenge effizient implementiert sein. Wie wir später sehen werden, entsprechen bei unserem Anwendungsfall alle Knoten aus der Start- bzw. Zielknotenmenge im Arbeitsgraphen jeweils genau einem Knoten im Ausgangsgraphen, so dass die Aufzählugn und der Test effizient realisierbar sein sollten.

3.1.4 Verträglichkeit mit Beschleunigungstechniken

Die Transformation muss für jede Suchanfrage separat geschehen. Das empfiehlt sich selbst dann, wenn die möglichen Start- und Zielknotenmengen vorher bekannt sind: Die Verwaltung dieser Knoten während der gesamten Lebensdauer des Graphen ist nur unnötiger Ballast. Das bringt jedoch eine kleine Einschränkung bei der Verwendung mit Beschleunigungstechniken mit sich: Manche dieser Techniken erwarten bestimmte Attribute bei Knoten oder Kanten, die bei den synthetisierten Knoten und Kanten natürlich nicht da sind. Diese Attribute werden bei den meist DIJKSTRA-basierten Algorithmen verwendet, um zu entscheiden, ob eine Kante besucht bzw. ein Knoten in die Prioritätswarteschlange aufgenommen werden soll.

Unsere Dummy-Knoten und -Kanten sollen aber auf jeden Fall besucht werden dürfen. Also müssen die Beschleunigungstechniken für die Verwendbarkeit mit der statischen Lösung des MSTSP dahingehend erweitert werden, dass das Fehlen von Knoten- oder Kantenattributen erlaubt ist und dazu führt, dass der Knoten oder die Kante in jedem Fall betrachtet werden.

Allerdings können nicht alle Beschleunigungstechniken mit dem Fehlen von Attributwerten zufrieden gestellt werden. Beispielsweise werden bei der zielgerichteten Suche explizit die Koordinaten des Zielknotens abgefragt, und die Suche wird daran ausgerichtet. Fehlen die Koordinaten, ist diese Beschleunigungstechnik wirklungslos. Da für die von uns betrachteten on-line-Verfahren zur Berücksichtigung der Abbiegeverbote gilt, dass die Start- und Zielknotenmengen im Arbeitsgraphen jeweils einem Knoten im Ausgangsgraphen entsprechen, können allen Knoten aus V_s bzw. V_t die gleichen Koordinaten zugeordnet werden, die jeweils auch für unsere Dummy-Knoten übernommen werden können.

Obige Ausführungen sollen nur als Beispiel dafür dienen, dass für verschiedene Beschleunigungstechniken unter Umständen verschiedene Strategien notwendig sind, um das Fehlen von Attributen zu kompensieren.

3.2 Dynamischer Ansatz für den Algorithmus von DIJKSTRA

3.2.1 Beschreibung

Durch eine leichte Anpassung kann der Algorithmus von DIJKSTRA das MSTSP lösen, ohne den Graphen zu transformieren. Dazu müssen nur die Zeilen 5 und 20 angepasst werden. Bei der Initialisierung von Q und C_s wird nunmehr die gesamte Startknotenmenge in die Warteschlange aufgenommen und C_s für jeden Knoten aus der Startknotenmenge auf Null zurückgesetzt. Der Algorithmus bricht ab, wenn der aus der Warteschlange entnommene Knoten in der Zielknotenmenge enthalten ist.

Im Anhang A ist auf Seite 41 der Pseudocode des angepassten Algorithmus dargestellt. Die Änderungen im Vergleich zum Standard-Algorithmus sind hellblau hervorgehoben.

3.2.2 Korrektheit

Wir betrachten, wie der Standard-Algorithmus auf dem modifizierten Graphen aus Abschnitt 3.1 abläuft. Im ersten Schritt wird v_s aus der Prioritätswarteschlange entnommen. Der Knoten v_s ist nicht Zielknoten, also werden alle Nachfolger von v_s (nämlich die Knoten aus V_s) zur Warteschlange

hinzugefügt. Da jede Kante von v_s zu einem Knoten aus V_s Kosten Null hat, haben die neu hinzugefügten Einträge zur Prioritätswarteschlange auch Kosten Null. Dieser Zustand entspricht genau dem, der nach Ausführung von Zeile 5 bei unserem modifizierten Algorithmus entsteht.

Wird ein Knoten v aus V_t aus der Prioritätswarteschlange entnommen, ist auch er nicht Zielknoten. Alle Nachfolger von v, darunter auch v_t, werden der Prioritätswarteschlange hinzugefügt. Der Knoten v_t erhält dabei dieselben Kosten wie v. In einem der nächsten Schritte wird v_t entnommen. Im Vergleich dazu bricht unser modifizierter Algorithmus bereits ab, wenn v aus der Prioritätswarteschlange entfernt wird. Die Länge des so ermittelten kürzesten Wegs ändert sich nicht: Die beim MSTSP ermittelten Wegkosten zu v_t entsprechen wegen $c_X((v, v_t)) = 0$ genau den Wegkosten zu v.

Folglich entspricht die Ausführung unseres modifizierten Algorithmus auf dem unveränderten Graphen genau der Ausführung des Original-Algorithmus auf dem um Start- und Zielknoten erweiterten Graphen. Also kann das MSTSP auch mit Hilfe der beschriebenen Modifikation gelöst werden. □

3.2.3 Komplexität

Ähnlich wie bei der statischen Variante müssen auch hier die Aufzählung von V_s und der Test auf Enthaltensein in V_t effizient ausführbar sein. Ansonsten ändert sich nichts am asymptotischen Aufwand des Algorithmus.

3.2.4 Anpassung eines gegebenen Algorithmus an das MSTSP

Bei einer praktischen Anwendung wird es darum gehen, eine beschleunigte Variante des Algorithmus von DIJKSTRA zu erweitern, so dass das MSTSP gelöst wird. Vom Prinzip her wird diese Erweiterung genauso aussehen wie die hier beschriebene Erweiterung: Anpassung der Initialisierung und des Abbruchkriteriums. Dabei können Probleme auftreten, beispielsweise muss bei der bidirektionalen Suche das kompliziertere Abbruchkriterium speziell an die nun vorhandenen Start- und Zielknotenmengen angepasst werden. Ferner arbeiten manche Beschleunigungstechniken wie die zielgerichtete Suche in Richtung eines konkreten Ziels. Zu prüfen wäre, ob diese jeweils auch an mehrere Ziele angepasst werden können.

4 Berücksichtigung von Abbiegeverboten beim STSP

In diesem Abschnitt stellen wir vier Verfahren der Kürzesten-Wege-Suche unter Berücksichtigung von Abbiegeverboten vor. Als Erstes stellen wir ein älteres statisches Verfahren vor. Wir zeigen danach zwei dynamische Verfahren und schließen mit einer Variante des in [Sch00] vorgestellten statischen Knotensplittings ab. Beginnen werden wir wieder mit Begriffsdefinitionen.

4.1 Formales

Wir bezeichnen $T \subset E \times E$ als die Menge der Abbiegeverbote: Für $(e_0, e_1) \in T$ ist das Betreten der Kante e_1 nicht gestattet, wenn direkt vorher die Kante e_0 durchlaufen wurde. e_0 heißt *Verbotseingangskante* (kurz: *Verbotseingang*), e_1 ist die *Verbotsausgangskante* (kurz: *Verbotsausgang*) des Abbiegeverbots. Für $(e_0, e_1) \in T$ gilt stets $target(e_0) = source(e_1)$.

Zusätzlich definieren wir zur Vereinfachung der Notation folgende Funktionen:

- *Menge der verbotenen Ausgangs-* und *Eingangskanten*: Für alle Kanten $e \in E$ definieren wir eine Funktion $forbidden_T : E \to 2^E$ mit $forbidden_T(e) := \{e' \in E : (e, e') \in T\}$. Analog definieren wir $forbidden^{-1}{}_T : E \to 2^E$ mit $forbidden^{-1}{}_T(e) := \{e' \in E : (e', e) \in T\}$. Ist T aus dem Kontext ersichtlich, lassen wir den Index weg. Die Funktionen werden auf natürliche Weise auf Kantenmengen erweitert.

- *Menge der erlaubten Ausgangs- und Eingangskanten*: Für alle Kanten $e \in E$ definieren wir eine Funktion $allowed_T : E \to 2^E$ mit $allowed_T(e) := out(target(e)) \setminus forbidden_T(e)$. Symmetrisch dazu definieren wir $allowed^{-1}{}_T : E \to 2^E$ mit $allowed^{-1}{}_T(e) := in(source(e)) \setminus forbidden^{-1}{}_T(e)$. Ist T aus dem Kontext ersichtlich, lassen wir den Index wieder weg. Die Funktionen werden ebenfalls auf natürliche Weise auf Kantenmengen erweitert.

4.2 Kantenaufnahme

4.2.1 Beschreibung

Als erstes statisches (den Algorithmus nicht veränderndes) Verfahren möchten wir die Kürzeste-Wege-Suche mit Kantenaufnahme vorstellen. Dieses Verfahren wurde zuerst in [SB77] vorgestellt. Obwohl es in [Sch00] als dynamisches Verfahren beschrieben wird, möchten wir hier eine statische Sichtweise zeigen.

Bei diesem Verfahren wird die STSP-Suchanfrage zu einem Graphen mit Abbiegeverboten in eine MSTSP-Suchanfrage auf einem Arbeitsgraphen ohne Abbiegeverbote umgeformt, indem die Kanten des Ausgangsgraphen zu Knoten im Arbeitsgraphen werden:

- $G_E(E, L, d)$: Arbeitsgraph mit Knotenmenge E, Kantenmenge L und Kostenfunktion $d : L \to \mathbb{R}_+$.
  - Knotenmenge E: Die Kanten des Ausgangsgraphen sind jetzt die Knoten unseres Arbeitsgraphen.
  - Kantenmenge $L := \{(e_0, e_1) \in E \times E : target(e_0) = source(e_1)\} \setminus T$: Sie enthält alle Kantenpaare des Ausgangsgraphen, die einen Pfad bilden, außer die durch ein Abbiegeverbot verbotenen Kantenpaare.
  - Kostenfunktion $d((e_0, e_1)) := c(e_0)$.
- E_s, E_t: Start- und Zielknotenmengen im Arbeitsgraphen. Wir wählen $E_s := out(v_s)$ und bestimmen eine möglichst kleine Menge $E_t \subseteq out(v_t)$ derart, dass $allowed^{-1}(E_t) = in(v_t)$ ist. Die Menge E_t ist wohldefiniert, da wir für jede Kante gefordert haben, dass sie betreten und verlassen werden kann: Zumindest für $E_t = out(v_t)$ gilt $allowed^{-1}(E_t) = in(v_t)$. Die Mengen E_s und E_t sind nichtleer, da wir für unsere Graphen starken Zusammenhang gefordert haben.

  Die Verwendung eines beliebigen Elements aus $out(v_t)$ als Zielknoten kann zu einer fehlerhaften Abbildung führen: Die Kante könnte Verbotsausgang eines Abbiegeverbots sein, das genau die Kante als Verbotseingang hat, über die der kürzeste Weg den Knoten v_t erreicht. Gibt es jedoch eine Kante e_t mit $forbidden^{-1}(e_t) = \emptyset$, können wir $E_t := \{e_t\}$ wählen.

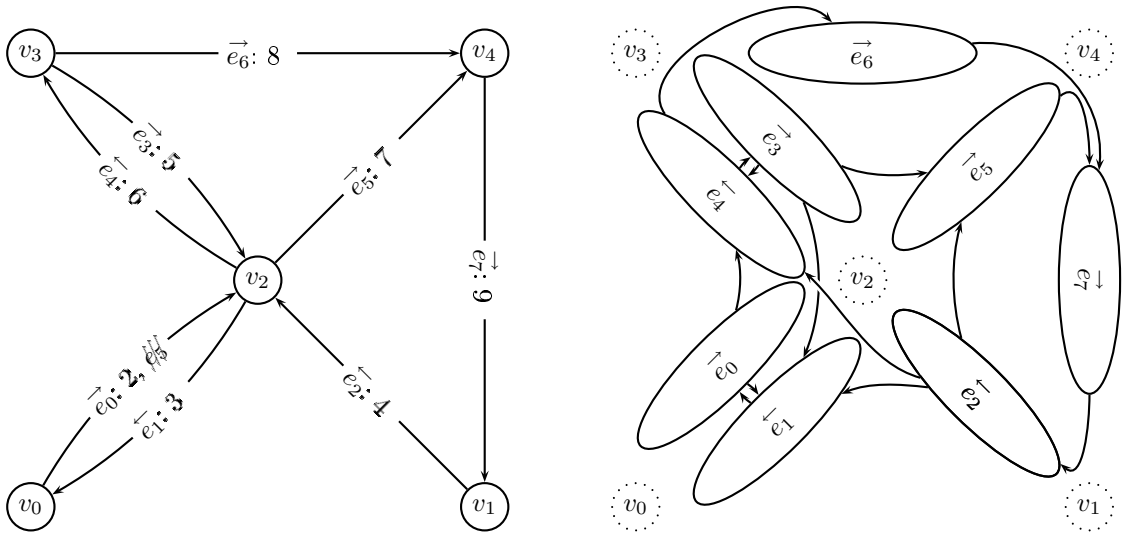
Abbildung 2 zeigt ein Beispiel für die Transformation eines Graphen. Der Ausgangsgraph ist stark zusammenhängend und enthält ein Abbiegeverbot (e_0, e_5). Das Abbiegeverbot spiegelt sich im Arbeitsgraphen als Fehlen einer Kante zwischen e_0 und e_5 wieder. Die Kantenkosten wurden im Arbeitsgraphen aus Gründen der Übersichtlichkeit nicht angegeben. Die Knoten v_0 bis v_5 (mit gepunkteten Linien) sind nur zur Orientierung angegeben und nicht Bestandteil des Arbeitsgraphen.

4.2.2 Korrektheit

Man sieht leicht, dass diese Umformung das Gewünschte erreicht. Jeder gültige Pfad $P_E = \langle l_1, \ldots, l_n \rangle$ in G_E entspricht eindeutig einem Pfad ohne Abbiegeverbote $P = \langle e_1, \ldots, e_n \rangle$ in G: Wir wählen $e_i := source(l_i)$ für alle $i \in \{1, \ldots, n\}$. Gilt zusätzlich $source(l_1) \in E_s$ und $target(l_n) \in E_t$, dann gilt auch $source(e_1) = v_s$ und $target(e_n) = v_t$.

Sei i beliebig aus $\{1, \ldots, n-1\}$. Wir zeigen zunächst, dass $l_i = (e_i, e_{i+1})$ gilt. Da per Definition $source(l_i) = e_i$ gilt, genügt zu zeigen, dass $target(l_i) = e_{i+1}$ ist:

$$P_E \text{ ist gültig} \quad \Leftrightarrow \quad target(l_i) = source(l_{i+1}) \quad \Big| \text{ Gültigkeit eines Pfads}$$



Ausgangsgraph $G(V, E, c)$ Arbeitsgraph $G_E(E, L, d)$

Abbildung 2: Beispiel für die Transformation eines Graphen bei der Kantenaufnahme

$$\Leftrightarrow \quad target(l_i) = e_{i+1} \qquad \text{Definition von } P$$

Der Pfad P ist gültig, da $target(e_i) = source(e_{i+1})$ ist. Er enthält keine Abbiegeverbote, da $(e_i, e_{i+1}) \notin T$ gilt. Beides ergibt sich sofort aus $(e_i, e_{i+1}) = l_i \in L$ und der Definition von L.
Für die Start- und Zielknoten von P_E gilt Folgendes:

$$\begin{aligned} source(l_1) \in E_s &\Leftrightarrow source(l_1) \in out(v_s) && \text{Definition von } E_s \\ &\Leftrightarrow e_1 \in out(v_s) && \text{Definition von } l_1 \\ &\Leftrightarrow source(e_1) = v_s && \text{Definition von } out \end{aligned}$$

$$\begin{aligned} target(l_n) \in E_t &\Rightarrow target(l_n) \in out(v_t) && E_t \subseteq out(v_t) \qquad (\dagger) \\ &\Leftrightarrow source(target(l_n)) = v_t && \text{Definition von } out \\ &\Leftrightarrow target(source(l_n)) = v_t && \text{Definition von } L \\ &\Leftrightarrow target(e_n) = v_t && \text{Definition von } l_n \end{aligned}$$

Für die Kosten der Pfade gilt:

$$\begin{aligned} d(P_E) &= \sum_{i=1}^{n} d(l_i) && \text{Definition Pfadkosten} \\ &= \sum_{i=1}^{n} c(e_i) && \text{Definition } d \\ &= c(P) && \text{Definition Pfadkosten} \end{aligned}$$

Umgekehrt existiert zu jedem gültigen Pfad P in G, der keine Abbiegeverbote enthält, (mindestens) ein entsprechender gleichlanger Pfad P_E in G_E. Die Konstruktion verläuft analog – wir wählen $l_i := (e_i, e_{i+1})$ für $i < n$ und $l_n := (e_n, e_t)$ mit e_t beliebig aus $E_t \cap \mathit{allowed}(e_n)$. Unsere Wahl von e_t ist immer möglich: Andernfalls gälte auch $e_n \notin \mathit{allowed}^{-1}(E_t)$, was wegen $e_n \in \mathit{in}(v_n)$ der Definition von E_t widerspricht. Der Korrektheitsnachweis für die Rückrichtung sieht ähnlich aus, wir verzichten hier auf dessen Darstellung.

Damit ist gezeigt, dass jeder Pfad in G_E von einem der Knoten aus E_s zu einem der Knoten aus E_t einem Pfad in G von v_s nach v_t ohne Abbiegeverbote entspricht, ferner dass zu jedem Pfad in G ohne Abbiegeverbote ein entsprechender Pfad in G_E existiert und dass die Pfadkosten jeweils gleich sind. Ein kürzester Weg in G_E zwischen allen Paaren von Start- und Zielknoten entspricht also einem kürzesten Weg in G unter Berücksichtigung der Abbiegeverbote. Wir können also jeden Kürzeste-Wege-Algorithmus zwischen Knotenpaaren benutzen, um die Anfrage nach dem kürzesten Weg unter Berücksichtigung der Abbiegeverbote zu beantworten. □

Im Übrigen ist die Pfad-Transformation bis auf die Wahl der Kante l_n in Gleichung (†) eineindeutig. Bei der Abbildung eines gegebenen Pfads in G auf einen Pfad in G_E gilt für l_n stets $\mathit{source}(l_n) = e_n$, aber $\mathit{target}(l_n)$ ist unter Umständen nicht eindeutig bestimmt. Trotzdem beeinflusst das die Kosten $d(P_E)$ nicht, da die Pfadkosten unabhängig von $\mathit{target}(l_n)$ sind.

Wir betrachten erneut unser Beispiel in Abbildung 2. Der kürzeste Weg zwischen v_0 und v_4 unter Berücksichtigung der Abbiegeverbote ist hier $\langle (v_0, v_2), (v_2, v_3), (v_3, v_4) \rangle = \langle e_0, e_4, e_6 \rangle$. Bei der Suche des kürzesten Wegs im Arbeitsgraphen haben wir $E_s = \{e_0\}$ und $E_t = \{e_7\}$ als Start- bzw. Zielknotenmengen, die einzigen einfachen Wege zwischen (einem Knoten aus) E_s und (einem Knoten aus) E_t sind $\langle (e_0, e_4), (e_4, e_6), (e_6, e_7) \rangle$ und $\langle (e_0, e_4), (e_4, e_3), (e_3, e_5), (e_5, e_7) \rangle$. Dabei ist der letztgenannte Weg teurer. Die Rücktransformation des erstgenannten Wegs mit $e_i := \mathit{source}(l_i)$ liefert $\langle e_0, e_4, e_6 \rangle$ als kürzesten Weg im Ausgangsgraphen unter Berücksichtigung der Abbiegeverbote.

4.2.3 Implementation

Um eine effiziente on-line-Implementation für dieses Verfahren zu erzeugen, müssen die in den Unterabschnitten 2.2.3 und 2.3.3 angegebenen Operationen effizient ausführbar sein. Unter anderem muss die Funktion *out* auf G_E effizient berechenbar sein. Das ist sie, wenn die Funktionen *out* für G und *forbidden* effizient ausgewertet werden können. Letztere kann als Array von kleinen Bitvektoren implementiert werden, wenn wir für die Knoten in G einen beschränkten Ausgangsgrad annehmen können – für Straßengraphen ist das sicher gegeben. Entsprechend können wir auch die anderen Zugriffsoperationen in der geforderten Art und Weise realisieren.

Für eine ausführlichere Diskussion zu diesem Thema, allerdings für ein anderes statisches Verfahren, verweisen wir auf den noch folgenden Unterabschnitt 4.5.5.

4.2.4 Komplexität

Der Graph G_E enthält die Kanten von G als Knoten und Kantenpaare von G als Kanten. Für dünne Graphen mit $|E| = O(|V|)$ und mit beschränktem Knotengrad (wie das bei Straßengraphen der Fall ist) ist G_E um einen konstanten Faktor größer als G, vergleiche dazu Abschnitt 6.1. Es ist also zu erwarten, dass die Laufzeit um einen konstanten Faktor steigt. Wir werden das empirisch überprüfen.

4.2.5 Bewertung

Die Transformation ist auch für den on-line-Fall sehr einfach zu implementieren. Bestehende Algorithmen für das MSTSP können ohne Anpassung verwendet werden.

Als Haupt-Nachteil wirkt sich der unnötig große Arbeitsgraph aus, dessen Knotenanzahl nicht von der Anzahl der Abbiegeverbote abhängt. Ein weiterer Nachteil ist die Notwendigkeit, mehrere Start- und Zielknoten verwalten zu müssen.

4.3 Mehrfache Knotenaufnahme

4.3.1 Motivation

Das Verfahren der Kantenaufnahme bläht den Arbeitsgraphen unnötig auf. Daher ist es sinnvoll, nach knotenbasierten Verfahren zu suchen, die den Suchraum nur soweit wie unbedingt nowendig erweitern.

Wir werden in diesem Abschnitt eine Abwandlung des DIJKSTRA-Algorithmus vorstellen, die Abbiegeverbote berücksichtigt – also ein dynamisches Verfahren. Der Algorithmus wurde in [Sch00] skizziert, allerdings nicht vollständig ausgeführt. Der Name rührt daher, dass es bei einer dynamischen Modifikation des Algorithmus in der Tat notwendig sein kann, bestimmte Knoten mehrfach in die Prioritätswarteschlange Q einzufügen, da sonst unter Umständen nicht der kürzeste Weg gefunden wird.

SCHMID erläutert in seiner Arbeit eine naive Abwandlung des Algorithmus von DIJKSTRA. Dort werden bei der Iteration über alle Ausgangskanten in Zeile 30 diejenigen Kanten einfach ausgelassen, die Verbotsausgang der aktuellen Vorgängerkante sind. Anschließend wird an einem einfachen Beispiel illustriert, dass dieses naive Verfahren nicht immer den kürzesten Pfad findet: Beispielsweise können keine Pfade gefunden werden, die einen Knoten mehrfach besuchen. Aber auch bei unserem Beispiel in Anhang B ab Seite 44 findet die naive Abwandlung des Algorithmus keinen Weg zwischen v_1 und v_4. Für eine ausführliche Diskussion hierzu verweisen wir auf [Sch00].

4.3.2 Formales

Zur Vereinfachung der Beschreibung verwenden wir die folgenden Operationen aus der relationalen Algebra [LL95]:

Selektion Sei $M \subseteq X_1 \times X_2 \times \ldots \times X_n$ eine Menge von n-Tupeln, die mit $(x_1, x_2, \ldots, x_n)$ bezeichnet sind. Sei ferner f ein Prädikat, das einige Variablen x_1, x_2 bis x_n als gebundene Variablen enthält. Dann enthalte die Menge $\sigma_f(M) \subseteq M$ die Elemente aus M, für die das Prädikat f erfüllt ist, wenn die gebundenen Variablen durch die entsprechenden Tupel-Indizes substituiert werden: $\sigma_f(M) := \{(x_1, x_2, \ldots, x_n) : f \text{ ist erfüllt}\}$.

Beispielsweise ist $\sigma_{r \geq 0}(\mathbb{R}) = \mathbb{R}_+$.

Projektion Sei $M \subseteq X_1 \times X_2 \times \ldots \times X_n$ wieder eine Menge von n-Tupeln, die mit $(x_1, x_2, \ldots, x_n)$ bezeichnet sind. Sei $i_1, \ldots, i_k$ $(k \leq n)$ eine Folge von paarweise verschiedenen Indizes mit $1 \leq i_j \leq n$. Dann soll $\pi_{i_1,\ldots,i_k}(M) \subseteq X_{i_1} \times \ldots \times X_{i_k}$ alle Tupel aus M mit den Indizes $i_1, \ldots, i_k$ enthalten. Es gilt also $\pi_{i_1,\ldots,i_k}(M) = \{(x_{i_1}, \ldots, x_{i_k}) : (x_1, x_2, \ldots, x_n) \in M\}$.

Es gilt zum Beispiel $\pi_2(\mathbb{R} \times \mathbb{R}_+) = \mathbb{R}_+$.

4.3.3 Beschreibung

Die Anpassung des Algorithmus besteht aus zwei Teilen. Zum einen wird die Wahl der betrachteten Folgekanten eingeschränkt, dazu wird die Domäne der Prioritätswarteschlange verändert. Zum zweiten verläuft die Aktualisierung der Prioritätswarteschlange komplizierter. Die Zeilennummern bei der Beschreibung beziehen sich auf den Pseudocode des Algorithmus im Anhang A auf Seite 42.

Auswahl der zu betrachtenden Kanten Einen Knoten mehrfach in die Prioritätswarteschlange aufzunehmen macht nur dann Sinn, wenn dieser Knoten Startknoten eines Verbotsausgangs ist. In diesem Fall darf dieser Knoten über bestimmte Kanten nicht verlassen werden.

Um bei der Abarbeitung eines Knotens v_{min} zu prüfen, über welche Kanten dieser verlassen weden darf, erweitern wir die Domäne der Prioritätswarteschlange. Sie soll neben den Knoten auch noch die Liste der erlaubten Kanten speichern, über die der Knoten verlassen werden darf.

Außerdem genügt jetzt das Vorgänger-Array P_s aus der Implementation ohne Abbiegeverbote nicht mehr, um den kürzesten Weg zu rekonstruieren: Es kann für einen Knoten in Abhängigkeit von der zu durchschreitenden Folgekante verschiedene Vorgängerkanten geben. Daher definieren

wir jetzt $P_s : E \to E$, so dass wir zu jeder Kante eindeutig die Vorgängerkante ermitteln können. Dadurch wird auch die Ermittlung des kürzesten Weges nach erfolgreicher Suche leicht verändert. Entsprechend benötigen wir bei jedem Eintrag in der Prioritätswarteschlange die Information, auf welche Vorgängerkante sich dieser Eintrag bezieht. Dazu erweitern wir wieder die Domäne der Prioritätswarteschlange, so dass auch die Vorgängerkante Bestandteil des Eintrags ist.

Insgesamt erhalten wir $Q \subset \mathbb{R}_+ \times (V \times 2^E \times E)$. Die Tupel bezeichnen wir mit $q = (c, (v, A, e))$ mit entsprechender Indizierung. Die Klammern dienen nur der Gruppierung, wir betrachten die Elemente von Q als 4-Tupel und wenden die Projektions-Operation entsprechend darauf an. Es gilt stets $v = \mathit{target}(e)$ und $A \subseteq \mathit{out}(v)$.

In jedem Schritt der Hauptschleife entfernen wir ein Element $q_{min} = (c_{min}, (v_{min}, A_{min}, e_{min}))$ aus Q. Die Prüfung, über welche Ausgangskanten v_{min} verlassen werden darf, geschieht in Zeile 30 des Algorithmus. Der Knoten darf nämlich genau über die Kanten aus A_{min} verlassen werden, wir iterieren also e_{neu} über alle A_{min}.

Für den weiteren Verlauf halten wir die Menge der bereits abgearbeiteten Kanten $E_{besucht}$ fest. Bei jedem aus Q entfernten Eintrag setzen wir $E_{besucht} := E_{besucht} \cup A_{min}$.

Sind wir am Ziel angelangt, ist die Vorgängerkante e_{min} des zuletzt extrahierten Eintrags unsere Ausgangskante für die Rückverfolgung. Wir setzen solange $e_{min} := P[e_{min}]$, bis $P[e_{min}]$ auf *nil* verweist. Die so ermittelte Folge der e_{min} ist unser kürzester Pfad in umgekehrter Reihenfolge der Kanten.

Aktualisierung der Prioritätswarteschlange Damit das Entfernen eines Elements aus Q und die Ermittlung der erlaubten Folgekanten so einfach erfolgen kann, muss im Gegenzug das Einfügen von Einträgen in die Prioritätswarteschlange mit einer gewissen Sorgfalt betrieben werden. Wir geben im Folgenden eine detaillierte Beschreibung der Einfügeprozedur an.

Dabei betrachten wir eine Iteration der inneren Schleife ab Zeile 30. Dort bestimmen wir, ob ausgehend von der Kante e_{neu} ein Eintrag zur Prioritätswarteschlange hinzugefügt wird und welche Kanten die Menge der erlaubten Ausgangskanten A_{neu} bei dem eventuell hinzugefügten Eintrag $q_{neu} = (c_{neu}, (v_{neu}, A_{neu}, e_{neu}))$ enthält. Es wird also im Voraus geprüft, welche Kanten in der Iteration der Hauptschleife erlaubt sind, bei der q_{neu} aus der Prioritätswarteschlange entfernt wird.

Zunächst definieren wir für jeden Knoten v_{neu} die Menge der zu diesem Knoten gehörenden Einträge aus Q:

$$Q_{v_{neu}} := \sigma_{v=v_{neu}}(Q).$$

Die Menge $Q_{v_{neu}}$ wird in zwei disjunkte Mengen $Q^-_{v_{neu}}$ und $Q^+_{v_{neu}}$ aufgeteilt, die jeweils die Elemente mit nicht höheren bzw. höheren Kosten als c_{neu} enthalten:

$$\begin{aligned} Q^-_{v_{neu}} &:= \sigma_{c \le c_{neu}}(Q_{v_{neu}}) \\ Q^+_{v_{neu}} &:= \sigma_{c > c_{neu}}(Q_{v_{neu}}) \quad = \quad Q_{v_{neu}} \setminus Q^-_{v_{neu}} \end{aligned}$$

Wir definieren drei Regeln, die angeben, wann das Verlassen des Knotens $v_{neu} = \mathit{target}(e_{neu})$ über eine Kante e untersagt ist:

1. Die Kante e ist Verbotsausgang eines Abbiegeverbots, und die Vorgängerkante in q_{neu} (also e_{neu}) ist Verbotseingang dieses Abbiegeverbots. In dem Fall darf e nicht betrachtet werden, da sonst eventuell ein kürzester Weg gefunden werden würde, der ein Abbiegeverbot enthält. Es muss also folgendes gelten:
$$e \notin \mathit{forbidden}(e_{neu}).$$

2. Die Kante e kann mit geringeren Kosten erreicht werden, weil ein Eintrag zu dem Knoten v_{neu}, bei dem das Verlassen über e erlaubt war, schon einmal aus der Prioritätswarteschlange

entfernt wurde. Die erneute Betrachtung von e liefert keinen kürzeren Weg, da die Kantengewichte nichtnegativ sind, und könnte außerdem zu einer Endlosschleife führen. Wir fordern folglich:

$$e \notin E_{besucht}.$$

3. Die Kante e kann mit geringeren Kosten erreicht werden, weil ein Eintrag zu dem Knoten v_{neu}, bei dem das Verlassen über e erlaubt ist, mit geringeren Kosten in der Prioritätswarteschlange vertreten ist. Die erneute Betrachtung von e führt zu keiner Endlosschleife, kann sich aber negativ auf die Laufzeit auswirken. Es ergibt sich folgende Forderung:

$$e \notin \bigcup \pi_3 \left(Q^-_{v_{neu}}\right).$$

Über die Kanten, die allen drei Regeln entsprechen, darf v_{neu} verlassen werden: Diese Kanten wurden noch nicht betrachtet, und der so aufgebaute kürzeste Weg enthält auch keine Abbiegeverbote. Also können wir die Menge der erlaubten Kanten wie folgt berechnen:

$$A_{neu} = out(v_{neu}) \setminus \left[forbidden(e_{neu}) \cup E_{besucht} \cup \left(\bigcup \pi_3 \left(Q^-_{v_{neu}}\right)\right)\right].$$

Durch das Hinzufügen des Eintrags $(c_{neu}, (v_{neu}, A_{neu}, e_{neu}))$ zu Q kann es passieren, dass bereits in Q befindliche Einträge zu v_{neu} mit höheren Kosten als c_{neu} (also Einträge in $Q^+_{v_{neu}}$) gegen Regel 3 verstoßen. Daher müssen alle Einträge in $Q^+_{v_{neu}}$ geprüft und gegebenenfalls angepasst werden. Um nicht mehr gegen Regel 3 zu verstoßen, müssen die Kanten in A_{neu} aus den entsprechenden Kantenlisten der Einträge in $Q^+_{v_{neu}}$ entfernt werden. Dazu definieren wir

$$Q^{++}_{v_{neu}} := \left\{(c, (v, A \setminus A_{neu}, e)) : (c, (v, A, e)) \in Q^+_{v_{neu}}\right\}$$

und ersetzen in Q alle Einträge aus $Q^+_{v_{neu}}$ durch die entsprechenden Einträge aus $Q^{++}_{v_{neu}}$.

Zur zusätzlichen Optimierung können wir die Einträge aus Q entfernen, die eine leere erlaubte Kantenmenge haben. Wir verzichten auf diese Optimierung, falls sich der Eintrag auf den Zielknoten v_t bezieht, um das Finden des kürzesten Wegs zum Zielknoten in keinem Fall zu beeinträchtigen. Diese Optimierung ändert nichts am Ablauf, da die auf diese Art und Weise entfernten Einträge beim Durchlauf der Hauptschleife ohnehin ignoriert werden würden.

Die Aktualisierung der Prioritätswarteschlange ergibt sich aus dem Hinzufügen des neuen Eintrags zu v_{neu}, aus der Aktualisierung der Einträge aus $Q^+_{v_{neu}}$ und aus dem Entfernen der Einträge mit leerer erlaubter Kantenmenge:

$$Q := \sigma_{A \neq \emptyset \vee v = v_t}\left(\left(Q \cup \{(c_{neu}, (v_{neu}, A_{neu}, e_{neu}))\} \setminus Q^+_{v_{neu}}\right) \cup Q^{++}_{v_{neu}}\right).$$

Man beachte, dass bei $A_{neu} = \emptyset$ und $v_{neu} \neq v_t$ der Inhalt der Prioritätswarteschlange nicht verändert wird, was vorab in Zeile 40 geprüft wird. Das entspricht genau dem Verzicht auf ein Einfügen eines „zu teuren“ Eintrags beim Algorithmus von DIJKSTRA ohne Abbiegeverbote.

Der gesamte Pseudocode des Algorithmus ist im Anhang A auf Seite 42 angegeben. Die Unterschiede zum Original-Algorithmus von DIJKSTRA sind wieder hervorgehoben. Da die durchgeführten Operationen vergleichsweise kompliziert sind, ist auch ein detailliertes Ablaufbeispiel des Algorithmus im Anhang B ab Seite 44 angegeben.

4.3.4 Korrektheit

Ein detaillierter Beweis des Algorithmus von DIJKSTRA ohne Abbiegeverbote ist in [CLRS01] zu finden. Dort wird eine Invariante definiert, die jeweils vor der Extraktion eines Elements aus der Prioritätswarteschlange in Zeile 11 gilt. Die Invariante gibt an, dass für jeden Knoten drei Zustände möglich sind:

Abgearbeitet Der kürzeste Weg ist bereits berechnet und kann über das Vorgänger-Array abgeleitet werden. Die Kosten des kürzesten Wegs zu allen abgearbeiteten Knoten sind nicht größer als der kleinste Kostenwert innerhalb der Prioritätswarteschlange.

Aufgrund der Nichtnegativität der Kantengewichte kann auch kein noch nicht abgearbeiteter Knoten mit geringeren Kosten als ein abgearbeiteter Knoten erreicht werden. Folglich bleiben die berechneten Kosten zu den abgearbeiteten Knoten für den restlichen Ablauf des Algorithmus unverändert, der berechnete kürzeste Weg zu einem abgearbeiteten Knoten ist endgültig.

In Bearbeitung Der Knoten wurde bereits besucht und befindet sich in der Prioritätswarteschlange. Der bisher bekannte kürzeste Weg kann ebenfalls über das Vorgänger-Array abgeleitet werden.

Hier kann es allerdings passieren, dass es einen kürzeren Weg zu einem Knoten „in Bearbeitung" gibt als bisher bekannt.

Noch nicht besucht Der Knoten wurde noch nicht besucht. Die Kosten zu dem Knoten sind noch nicht bekannt, möglicherweise existiert gar kein Weg zu diesem Knoten.

Bei jeder Extraktion eines Knotens aus der Warteschlange wechselt dieser Knoten aus dem Zustand „In Bearbeitung" in den Zustand „Abgearbeitet", und manche Knoten wechseln aus dem Zustand „Noch nicht besucht" in den Zustand „In Bearbeitung". Alle Zustandsübergänge erfolgen unter Beibehaltung der Invariante.

Der Beweis in [CLRS01] nimmt an, es gäbe einen kürzeren Weg als den berechneten. Es wird die erste Stelle inspiziert, an der sich der berechnete vom tatsächlichen kürzesten Weg unterscheidet. Durch Untersuchung des Ablauf des Algorithmus wird nachgewiesen, dass der tatsächliche kürzeste Weg mindestens genausolang (und damit auch gleichlang) wie der berechnete ist.

Durch die Überschaubarkeit der Aktualisierungs-Operation der Prioritätswarteschlange kann beim „einfachen" Algorithmus von DIJKSTRA vergleichsweise leicht der Invarianten-Nachweis geführt werden. Wir verzichten hier auf den sehr technischen Beweis, der ganz ähnlich aufgebaut ist, allerdings eine kompliziertere Invariante beinhaltet: Wir benötigen auch „teilweise abgearbeitete" Knoten, wenn ein Knoten über eine Kante erreicht wurde, von der Abbiegeverbote ausgehen. □

4.3.5 Implementation

Wir deuten im Folgenden an, wie die Aktualisierungs-Operation effizient implementiert werden kann.

Kantenmengen Zunächst bemerken wir, dass für jedes Tupel $(c, (v, A, e))$ in der Prioritätswarteschlange die Relation $A \subseteq out(v)$ gilt. Wenn wir beschränkten Ausgangsgrad annehmen, was bei Straßengraphen gegeben ist, können wir die Kantenmengen als kleine Bitvektoren implementieren, die in ein Maschinenwort passen.

Die Prüfung auf Enthaltensein und die Mengenoperationen Vereinigung, Schnitt und Subtraktion sind damit in $O(1)$ durchführbar und benötigen nur sehr wenige Maschinenbefehle.

Eine ähnliche Vorgehensweise bietet sich für die Implementation der Funktion *forbidden* und der Menge $E_{besucht}$ an: Beides kann als Array von kleinen Bitvektoren realisiert werden.

Alle zu einem Knoten gehörenden Einträge Um die Menge $Q_{v_{neu}}$ effizient berechnen zu können, kann redundant zur Prioritätswarteschlange ein Array von Listen von Zeigern auf Prioritätswarteschlangen-Einträge gepflegt werden. Das Pflegen dieses Arrays von Listen kostet nur einen konstanten Faktor an Laufzeit und Speicherplatz.

In unserer Referenz-Implementation wurde eine andere Strategie gewählt. Wir speichern von vornherein die Q_v in der Prioritätswarteschlange, wobei wir sortierte Listen verwenden, die aufsteigend nach Kosten sortiert. Bei jedem Entfernen eines Eintrags zu einem Knoten v_{min} aus der Warteschlange entfernen wir in Wirklichkeit die Liste $Q_{v_{min}}$ aller Einträge zu diesem Knoten. Aus dieser Liste entfernen wir wiederum das erste Element (das mit den geringsten Kosten) und fügen die übrig bleibende Liste, falls sie nicht leer ist, wieder in Q ein. Die Kosten einer Knotenliste sind als die Kosten des ersten Elements definiert.

Dadurch reduziert sich die Aktualisierungs-Operation auf einen Durchlauf einer sortierten Liste und das optionale Einfügen eines Elements während des Durchlaufs (ohne vorhergehende Suche). Da die Mengenoperationen in $O(1)$ realisiert werden können, benötigen wir eine Zeit linear in der Größe der Liste – das ist optimal für diesen Algorithmus.

In der inneren Schleife müssen wir $Q_{v_{neu}}$ zu einem gegebenen v_{neu} effizient ermitteln können. Hierfür genügt jetzt ein Array von Zeigern auf Prioritätswarteschlangen-Einträge, was im Vergleich zum naiven Ansatz Speicher spart.

4.3.6 Komplexität

Wir betrachten nur die Laufzeit für dünne Graphen mit $|E| = O(|V|)$ und beschränktem Knotengrad $deg = O(1)$.

Zu einem Knoten v können höchstens $indeg(v)$ verschiedene Einträge aus der Prioritätswarteschlange entfernt werden, da keine zwei Einträge mit derselben Vorgängerkante eingefügt werden. Folglich wird die äußere Schleife höchstens $\sum_{v \in V} indeg(v) = |E|$ Mal durchlaufen. Eine genauere Betrachtung liefert, dass jedes Abbiegeverbot zu höchstens einem zusätzlichen Eintrag in der Prioritätswarteschlange beiträgt. Wir können also die Anzahl der Durchläufe mit $|V|+|T|$ beschränken, was bei den meisten Anwendungsfällen kleiner sein wird als $|E|$.

Unsere Implementation benötigt in jedem Durchlauf der äußeren Schleife eine *delete_min*- und eine *insert*-Operation auf Q. Wird ein Fibonacci-Heap verwendet, dominiert *delete_min* mit $O(\log |Q|) = O(\log(|V| + |T|))$ die Kosten.

In der inneren Schleife kann jede Kante insgesamt höchstens einmal als e_{neu} verwendet werden. Bei jedem Durchlauf der inneren Schleife wird eine Liste, die höchstens deg Einträge enthält, durchlaufen. Zusätzlich kann ein *decrease_key* oder ein *insert* notwendig sein. Für $deg = O(1)$ erhalten wir $O(1)$ als Kosten für einen Durchlauf der inneren Schleife.

Insgesamt erhalten wir einen Laufzeitaufwand von $O(\min(|E|, |V| + |T|) \cdot \log(|V| + |T|)) \cong O((n + |T|) \cdot \log(n + |T|))$ im schlimmsten Fall, was etwas schlechter ist als beim „einfachen“ Algorithmus von DIJKSTRA.

Der benötigte Arbeitsspeicher wird analog zum Algorithmus von DIJKSTRA durch die maximale Größe der Prioritätswarteschlange angegeben, die wiederum durch die Anzahl Durchläufe der äußeren Schleife beschränkt ist. Wir erhalten also einen maximalen Speicheraufwand von $O(\min(|E|, |V| + |T|)) \cong O(n + |T|)$.

4.3.7 Bewertung

Der Algorithmus benötigt eine geringere Laufzeit als die Kantenaufnahme, wenn die Anzahl der Abbiegeverbote vergleichsweise gering ist. Dafür wird allerdings der Preis einer komplexen Aktualisierungs-Prozedur für die Prioritätswarteschlange gezahlt.

4.4 Knoten- und Kantenaufnahme

4.4.1 Motivation und Herleitung

Der in Abschnitt 4.3 vorgestellte Algorithmus arbeitet besser als die Kantenaufnahme, ist allerdings wesentlich komplizierter. Wir werden in diesem Abschnitt versuchen, den Algorithmus schrittweise unter Beibehaltung der Laufzeit zu vereinfachen.

Verzicht auf Regel 3 Zunächst betrachten wir Regel 3 bei der Mehrfach-Knotenaufnahme. Wenn diese Regel nicht eingehalten wird, kann es Kanten geben, die bei mehr als einem Eintrag der Prioritätswarteschlange in der erlaubten Kantenmenge vertreten sind:

$$\bigcap \pi_3(Q) \neq \emptyset.$$

Wir betrachten eine solche Kante $e_d \in \bigcap \pi_3(Q)$. Für diese Kante gibt es zwei Einträge $q_1 = (c_1, (v_1, A_1, e_1))$ und $q_2 = (c_2, (v_2, A_2, e_2))$, so dass $e_d \in A_1 \cap A_2$ gilt. Wegen $A_1 \subseteq out(v_1)$ und

$A_2 \subseteq \mathit{out}(v_2)$ gilt $v_1 = \mathit{source}(e_d)$ und $v_2 = \mathit{source}(e_d)$, also $v_1 = v_2$. Demnach ist auch $\mathit{target}(e_1) = \mathit{target}(e_2)$. Wir setzen jedoch nach wie vor voraus, dass jede Kante nur einmal in der inneren Schleife betrachtet wird, dass also $e_1 \neq e_2$ gilt.

O.B.d.A. nehmen wir an, dass der Eintrag q_1 vor dem Eintrag q_2 in der Prioritätswarteschlange ist, also $c_1 \leq c_2$ gilt. Nach Extraktion des Eintrags q_1 wird A_1 (und damit auch e_d) zur Menge der bearbeiteten Knoten $E_{besucht}$ hinzugefügt und anschließend auch in der inneren Schleife abgearbeitet. Wenn q_2 aus der Prioritätswarteschlange entfernt wird, ist also e_d bereits abgearbeitet und in $E_{besucht}$ enthalten. Nach Voraussetzung dürfen wir e_d nicht noch einmal abarbeiten.

Insbesondere gilt $e_d \in A_2 \cap E_{besucht}$ vor der Aktualisierung von $E_{besucht}$ in Zeile 27. Wenn wir also die Kanten in A_2 weglassen, die bereits als besucht markiert sind, lassen wir auch e_d aus. Andererseits sind nur solche Kanten in $A_{min} \cap E_{besucht}$, die gegen Regel 3 verstoßen. Durch Einfügen der Zuweisung

$$A_{min} \leftarrow A_{min} \setminus E_{besucht}$$

vor Zeile 27 erreichen wir genau das gewünschte Verhalten. Wird $E_{besucht}$ wie bei der Mehrfach-Knotenaufnahme als Array von kleinen Bitvektoren implementiert, kann diese Zuweisung in $O(1)$ ausgeführt werden.

Die Anzahl der Elemente in der Prioritätswarteschlange kann zwischenzeitlich größer werden als bei der Mehrfach-Knotenaufnahme. Sie wird jedoch die Kantenanzahl $|E|$ nie übersteigen. Trotzdem beeinflusst das die Laufzeit negativ. Wir werden diese Verschlechterung der Laufzeit später ausgleichen. Wenn die Kantenmengen als kleine Bitvektoren implementiert werden, entsteht durch die möglicherweise höhere Anzahl Kanten in den einzelnen Kantenmengen asymptotisch kein Laufzeit- oder Speichernachteil.

Verzicht auf Regel 2 Die Menge $E_{besucht}$ der besuchten Kanten vergrößert sich während der Abarbeitung unter Inklusion: Es werden stets nur neue Elemente hinzugefügt. Wenn eine Kante zum Zeitpunkt der Ausführung von Regel 2 in $E_{besucht}$ vor Einfügen des entsprechenden Eintrags q enthalten ist, so ist sie das sicher auch zum Zeitpunkt der Extraktion von q. Daher können wir ohne Weiteres auch auf Regel 2 komplett verzichten, wenn wir bei der Iteration in der innere Schleife die besuchten Kanten auslassen.

Auch hier werden höchstens die Kantenmengen A größer, was sich für kleine Bitvektoren aber nicht auf die asymptotische Laufzeit oder auf den Speicherbedarf auswirkt.

Verzicht auf die Menge der erlaubten Kanten Aus unserem Regelsatz ist nur eine Regel übrig geblieben, die angibt, dass Verbotsausgangskanten nicht erlaubt sind. Wird diese Regel eingehalten, gilt für alle Elemente $q = (c, (v, A, e))$ die Gleichung

$$A = \mathit{out}(v) \setminus \mathit{forbidden}(e).$$

Es genügt also, die Menge A der erlaubten Kanten erst nach der Extraktion des jeweiligen Elements aus der Prioritätswarteschlange zu berechnen, da sich weder *out* noch *forbidden* während der Abarbeitung des Algorithmus ändern. Daher müssen wir die Menge der erlaubten Kanten nicht mehr explizit mitspeichern und können $Q \subset \mathbb{R}_+ \times (V \times E)$ wählen.

An der asymptotischen Laufzeit ändert sich durch diese Sparmaßnahme nichts. Wir benötigen zwar weniger Speicher für die Prioritätswarteschlange, allerdings sparen wir nur einen konstanten Faktor.

Reduzierung der Anzahl betrachteter Elemente Man beachte, dass das derart modifizierte Verfahren im Wesentlichen eine dynamische Version der Kantenaufnahme ist. Die Beschleunigung der Mehrfach-Knotenaufnahme ist beim Verzicht auf Regel 3 wieder verloren gegangen. Wir werden diesen Nachteil ausgleichen.

Wir definieren zunächst eine Äquivalenzrelation $\equiv$ für Kanten wie folgt:

$$e_1 \equiv e_2 \iff e_1 = e_2 \vee (\mathit{target}(e_1) = \mathit{target}(e_2) \wedge \mathit{forbidden}(e_1) = \mathit{forbidden}(e_2) = \emptyset).$$

Kanten mit nichtleerer Verbotskantenmenge bilden jeweils eine eigene Äquivalenzklasse, Kanten mit leerer Verbotskantenmenge und gleichem Zielknoten werden zu einer Äquivalenzklasse zusammenfasst.

Die Äquivalenzrelation hat folgende Eigenschaft: Nachdem ein Prioritätswarteschlangen-Eintrag $q_1 = (c_1, (v, e_1))$ abgearbeitet wurde, sind bei der Abarbeitung eines weiteren Prioritätswarteschlangen-Eintrags $q_2 = (c_2, (v, e_2))$ mit $c_1 \leq c_2$ und $e_1 \equiv e_2$ alle Kanten aus $\mathit{out}(v) \setminus \mathit{forbidden}(e_2)$ bereits in $E_{besucht}$ enthalten. Das ist offensichtlich für die Kanten e_2 mit nichtleerer Verbotsmenge, da hier die entsprechende Äquivalenzklasse nur ein Element enthält und folglich $e_1 = e_2$ ist. (Unabhängig davon, dass zwei verschiedene Prioritätswarteschlange-Einträge mit gleicher Vorgängerkante gar nicht auftreten können.) Für Kanten e_2 mit leerer Verbotsmenge stellen wir fest, dass nach Definition der Äquivalenzrelation auch $\mathit{forbidden}(e_1) = \emptyset$ gilt. Daher wurden nach Abarbeiten des ersten Eintrags q_1 alle Kanten aus $\mathit{out}(v)$ in der inneren Schleife abgearbeitet – während der Abarbeitung von q_1 oder auch bereits vorher.

Aus diesem Grund genügt es, zu jedem Zeitpunkt höchstens einen Repräsentanten bezüglich der Äquivalenzrelation in der Prioritätswarteschlange zuzulassen. Selbstverständlich muss das derjenige mit den geringsten Wegkosten sein. Nach Abarbeitung eines Elements brauchen äquivalente Elemente auch nicht mehr in die Prioritätswarteschlange eingefügt zu werden.

Wir ändern nun unseren Algorithmus derart, dass es für jede Äquivalenzklasse höchstens einen Eintrag in der Prioritätswarteschlange geben kann. Dazu ändern wir wieder die Domäne der Prioritätswarteschlange, diesmal auf $Q \subset \mathbb{R}_+ \times (V \times E/_{\equiv})$. Beim Einfügen eines neuen Elements prüfen wir, ob ein Element mit einer äquivalenten Vorgängerkante bereits in der Prioritätswarteschlange vorhanden ist. Gibt es einen solchen Eintrag, wird wie bei der Original-Implementation der Eintrag mit den höheren Kosten verworfen. Analog fügen wir beim Aktualisieren der Menge $E_{besucht}$ zu jeder einzufügenden Kante e gleich alle Elemente ihrer Äquivalenzklasse $[e]_{\equiv}$ ein.

Dadurch reduziert sich die maximale Anzahl Durchläufe auf den Index der Relation $\equiv$ (auf die Anzahl deren Äquivalenzklassen), die wir wieder durch $\min(|E|, |V| + |T|)$ nach oben abschätzen können. Damit erreichen wir wieder dieselbe Laufzeit wie bei der Mehrfach-Knotenaufnahme. Der asymptotische Speicherbedarf ändert sich auch hier nicht.

Effiziente Implementation der Quotientenmenge Die soeben besprochene Änderung sieht vor, bei jeder Aktualisierung der Prioritätswarteschlange zu prüfen, ob ein Element mit einer äquivalenten Vorgängerkante bereits eingefügt wurde. Diese Prüfung kann mit Hilfe eines redundanten Arrays geschehen, das zu jeder Kante einen jeweils eindeutigen Repräsentanten speichert. Wir möchten auch hierfür einen alternativen Ansatz vorstellen.

Dazu definieren wir eine Menge $M := V \uplus E$ und eine Funktion h von E auf M:

$$h(e) := \begin{cases} \mathit{target}(e) : & \mathit{forbidden}(e) = \emptyset \\ e : & \text{sonst} \end{cases}.$$

Diese Funktion bildet äquivalente Kanten auf denselben Wert ab. Kanten mit $\mathit{forbidden}(e) = \emptyset$ sind äquivalent, wenn $\mathit{target}(e)$ jeweils auf denselben Knoten verweist, solche Kanten werden durch h auf den (gemeinsamen) Zielknoten abgebildet. Die anderen Kanten sind jeweils nur zu sich selbst äquivalent und werden auch auf sich selbst abgebildet. Man vergleiche dazu die Definition von $\equiv$:

In diesem Sinn ist h ein Isomorphismus zwischen $(E, \equiv)$ und $(M, =)$, es gilt

$$h(e_1) = h(e_2) \iff [e_1]_{\equiv} = [e_2]_{\equiv} \iff e_1 \equiv e_2.$$

Dadurch können wir $Q \subset \mathbb{R}_+ \times (V \times M)$ wählen, wenn wir beim Einfügen eines Elements die Funktion h und bei der Extraktion die Umkehrfunktion h^{-1} anwenden. Das hat den Vorteil, dass die Prüfung auf Existenz eines äquivalenten Elements in der Prioritätswarteschlange keine zusätzliche Datenstruktur mehr braucht.

Die Umkehrfunktion h^{-1} ist die Identitätsfunktion für Argumente aus E. Allerdings kann die Umkehrfunktion $h^{-1}(m)$ für $m \in V$ ohne Zusatzinformation nicht eindeutig berechnet werden: Wir benötigen noch die Kante, über die der Knoten m betreten wurde. Diese Vorgängerkante brauchen wir jedoch nur zur eindeutigen Bestimmung des Pfadverlaufs: Wir erinnern uns, dass

das Vorgängerarray P_s bei der Mehrfach-Knotenaufnahme als Abbildung $E \to E$ definiert wurde. (Für die Wahl der erlaubten Zielkanten wissen wir für $m \in V$, dass alle Ausgangskanten erlaubt sind, und benötigen die Vorgängerkante an dieser Stelle nicht.)

Wir lösen das Problem, indem P_s als Abbildung $M \to M$ definiert wird. Ferner definieren wir die Funktion $k : M \to V$ als

$$k(m) := \begin{cases} m & : m \in V \\ target(m) & : m \in E \end{cases} .$$

Beim Einfügen wenden wir h an. Bei der späteren Auswertung während der Pfadrekonstruktion verwenden wir die Kanten, die zwischen den Knoten $k(P_s[m])$ und $k(m)$ gezogen sind. Die Funktion k bildet die Elemente aus M auf deren zugehörige Knoten ab, so dass $k(h([e]_{\equiv})) = target(e)$ gilt. Da wir keine Multikanten zugelassen haben, ist die Vorgängerkante so eindeutig bestimmt. Wir benötigen die Umkehrfunktion also gar nicht.

Die Funktion h kann in $O(1)$ berechnet werden, deshalb ändern sich nach diesem Schritt weder Laufzeit noch Speicherbedarf des Algorithmus.

Um effizient ermitteln zu können, ob es in der Prioritätswarteschlange ein äquivalentes Element mit höheren Kosten gibt, pflegen wir ähnlich wie beim Standard-Algorithmus von Dijkstra ein Array $C_s : M \to \mathbb{R}_+$, das die niedrigsten bisher bekannten „Kosten“ zu den Elementen aus M angibt. Dabei verwenden wir auch hier h, um Äquivalenzklassen von Kanten auf die Elemente aus M abzubilden. Das hat den zusätzlichen Nebeneffekt, dass wir auf $E_{besucht}$ verzichten können: Eine bereits besuchte Kante wird dann zwar in der inneren Schleife abgearbeitet, das zugehörige $C_s[m_{neu}]$ ist jedoch kleiner als die gerade ermittelten Kosten, so dass der Eintrag zu m_{neu} nicht eingefügt wird.

Abschließend stellen wir fest, dass nunmehr für einen Eintrag $(c, (v, m))$ stets der Zusammenhang $v = k(m)$ gilt. Daher können wir auf die Speicherung der Knoten-Komponente verzichten und $Q \subset \mathbb{R}_+ \times M$ wählen.

4.4.2 Beschreibung

Nachdem wir im letzten Unterabschnitt den Algorithmus sehr ausführlich hergeleitet haben, stellen wir hier das Resultat vor. Im Anhang A ist auf Seite 43 der modifizierte Algorithmus als Pseudocode zu finden, ebenfalls für den Fall STSP. Auch hier sind die Unterschiede zum Original-Algorithmus von Dijkstra hervorgehoben.

Wir verwenden die oben beschriebene Menge $M := V \uplus E$ als Domäne der Prioritätswarteschlange, es ist also $Q \subset \mathbb{R}_+ \times M$. Zusätzlich zu Knoten aus V darf diese jetzt also auch Kanten aus E enthalten. Wie bei der Herleitung dargestellt, werden jedoch nur die Kanten jemals als Element eingefügt, die Ausgangspunkt eines Abbiegeverbots sind.

Bei der Bearbeitung eines Knotens v_{neu} fügen wir $h(e_{neu})$ in die Prioritätswarteschlange ein. Bei der Extraktion des minimalen Elements m_{min} aus Q erhalten wir $v_{min} = k(m_{min})$. Dazu gibt es zwei Fälle:

- $m_{min} \in V$. Das bedeutet, die Vorgängerkante definiert kein Abbiegeverbot, und alle Ausgangskanten sind erreichbar.

- $m_{min} \in E$. Hier ist m_{min} die Vorgängerkante, die Ausgangspunkt eines oder mehrerer Abbiegeverbote ist. In diesem Fall dürfen wir die Kanten nicht betrachten, die aufgrund eines Abbiegeverbots ausgehend von der Vorgängerkante m_{min} nicht betreten werden dürfen.

Analog zu Q setzen wir auch die Domänen von C_s und P_s auf M. Das ist notwendig, da ein Knoten in Abhängigkeit des zu erreichenden Ziels mehrere Vorgänger haben kann und diese unterschieden werden müssen.

Da ein eventuelles Abbiegeverbot keine Rolle mehr spielt, wenn beim Betreten einer Kante das Ziel erreicht wird, brechen wir die Suche auch bei Extraktion eines $m_{min} \in E$ mit $target(m_{min}) = v_t$ ab – also wenn $v_{min} = v_t$ ist.

4.4.3 Korrektheit

Der Algorithmus ist durch schrittweise Transformation der Mehrfach-Knotenaufnahme entstanden. Wir haben die Umwandlung ausführlich beschrieben und jeden Schritt begründet. Daraus können wir folgern, dass die Knoten- und Kantenaufnahme korrekt arbeitet, wenn wir von der Korrektheit der Mehrfach-Knotenaufnahme ausgehen können. □

4.4.4 Implementation

Der einzige Unterschied zum Algorithmus von DIJKSTRA ist die Domäne der Prioritätswarteschlange. Aus Effizienzgründen sollte versucht werden, die Elemente von M in ein Maschinenwort einzupassen. Ansonsten gibt es keine nennenswerten Implementationsdetails.

4.4.5 Komplexität

Wie bei der Herleitung bereits erwähnt, entspricht die Laufzeit der Knoten- und Kantenaufnahme der der Mehrfach-Knotenaufnahme, also $O(\min(|E|, |V| + |T|) \cdot \log(|V| + |T|)) \cong O((n + |T|) \cdot \log(n + |T|))$. Der Speicherbedarf verringert sich um einen konstanten Faktor, bleibt jedoch bei $O(\min(|E|, |V| + |T|)) \cong O(n + |T|)$.

4.4.6 Bewertung

Es ist uns gelungen, die vergleichsweise komplizierte Mehrfach-Knotenaufnahme unter Beibehaltung von Laufzeit und Speicheraufwand zu vereinfachen. Allerdings wurde auch hier der Algorithmus verändert, was den Einsatz mit Beschleunigungstechniken zumindest erschwert.

4.5 Knotensplitting

4.5.1 Motivation und Herleitung

Die letzten beiden vorgestellten Verfahren waren dynamisch, dabei wurde der Algorithmus verändert. Wir werden in diesem Abschnitt basierend auf der Knoten- und Kantenaufnahme ein statisches Verfahren herleiten und zeigen, wie es effizient als on-line-Verfahren implementiert werden kann. Der Name „Knotensplitting“ rührt daher, dass es bei diesem Verfahren zu einem Knoten im Ausgangsgraphen unter Umständen mehrere Knoten im Arbeitsgraphen geben kann und dazu der Knoten des Ausgangsgraphen quasi „aufgeteilt“ wird.

Aus dem Algorithmus von DIJKSTRA haben wir die Mehrfach-Knotenaufnahme und daraus die Knoten- und Kantenaufnahme entwickelt. Alle drei Verfahren haben gemeinsam, dass sie auf dem Graphen $G(V, E, c)$ arbeiten. Die letzten beiden Verfahren betrachten die Abbiegeverbote T zusätzlich zum Graphen.

Für die Knoten- und Kantenaufnahme ist auch eine andere Sichtweise naheliegend: Der Algorithmus arbeitet auf einem (bisher nicht genau spezifizierten) Graphen mit Knotenmenge M. In der Tat bestehen viele Unterschiedsmerkmale im Pseudocode daraus, dass ein v oder V durch ein m bzw. M ersetzt wird. Wir werden im Folgenden ähnlich wie in Abschnitt 4.4 die Knoten- und Kantenaufnahme transformieren. Das Ziel ist, unter Beibehaltung der Funktionalität den Algorithmus von DIJKSTRA zur Lösung des MSTSP als Ergebnis herauszubekommen – freilich arbeitet dieser nicht auf G, sondern auf einem transformierten Graphen. Das entspricht der Definition eines statischen Verfahrens. Wir beziehen uns bei den Zeilennummern auf den Pseudocode zur Knoten- und Kantenaufnahme im Anhang A auf Seite 43.

Berechnung der potentiell einzufügenden Elemente Zur Auswahl der betrachteten Ausgangskanten e_{neu} in Zeile 30 trägt die Menge F bei, die in den Zeilen 28 bzw. 29 festgelegt wird: Über Kanten aus F darf der aktuelle Knoten v_{min} nicht verlassen werden. Das einzufügende Element m_{neu} wird in Zeile 33 bestimmt: Je nachdem ob e_{neu} Eingangskante eines Abbiegeverbots ist, wird entweder der Knoten v_{neu} oder e_{neu} selbst eingefügt.

Wichtig ist hier, dass diese Berechnung entkoppelt von der eigentlichen Suche abläuft. Welche Elemente hinzugefügt werden, hängt (abgesehen vom Graphen und von den Abbiegeverboten) nur von m_{min} ab. Wir können also eine Funktion $out_M : M \to 2^{M \times M}$ definieren, so dass

$$out_M(m_{min}) = \{(m_{min}, m) : m \in M,\ m_{neu} \text{ nimmt in der inneren Schleife den Wert } m \text{ an } \}$$

gilt. Diese Funktion berechnet eine Menge von Paaren von Elementen aus M, wobei die erste Komponente immer das Argument und die zweite Komponente eben einen möglichen Wert für m_{neu} angibt. Die explizite Darstellung für out_M folgt:

$$out_M(m) := \begin{cases} \{(m, h(e)) : e \in out(k(m))\} & : m \in V \\ \{(m, h(e)) : e \in out(k(m)) \setminus forbidden(m)\} & : m \in E \end{cases}.$$

Wir stellen fest, dass es für $(m, m') \in out_M(m)$ ein $e \in out(k(m))$ geben muss, so dass $m' = h(e)$ gilt.

Wir bemerken ferner, dass in der inneren Schleife die Kante e_{neu} nur in den Zeilen 32 und 33 benötigt wird. Es gilt $e_{neu} = (k(m_{min}), k(m_{neu}))$, so dass wir die in Zeile 32 benötigte Kostenfunktion mit Hilfe der Funktion $c_M : M \times M \to \mathbb{R}_+$, die als

$$c_M(m_1, m_2) := c(k(m_1), k(m_2))$$

definiert ist, berechnen können. Der (letzte übrig bleibende) Zugriff in Zeile 33 entfällt, wenn wir auf e_{neu} ganz verzichten und statt dessen gleich über die (m_{min}, m_{neu}) aus $out_M(m_{min})$ iterieren. Das entspricht der Iteration über die Ausgangskanten eines Knotens beim Original-Algorithmus. Wir werden später die Kantenmenge des Arbeitsgraphen so wählen, dass die Ausgangskantenmenge zu einem m genau $out_M(m)$ ist.

Initialisierung und Abbruch Die Initialisierung in Zeile 5 und die Abbruchbedingung in Zeile 20 arbeiten noch mit v_s bzw. v_t als Start- bzw. Zielknoten. Dem Zielknoten entsprechen allerdings unter Umständen mehrere Elemente aus M. Hier wechseln wir zu dem bereits beschriebenen dynamischen Algorithmus zur Lösung des MSTSP und geben Start- und Ziel-Element-Mengen $M_s, M_t \subset M$ an.

Bei der Initialisierung wurde v_s in die Prioritätswarteschlange eingefügt, daher können wir $M_s := \{v_s\}$ setzen. Der Zielknoten ist erreicht, wenn $k(m_{min}) = v_{min} = v_t$ ist. Das ist genau dann der Fall, wenn $m_{min} \in \{v_t\} \uplus in(v_t)$ ist. Wir wählen also

$$M_t := \{m \in M : k(m) = v_t\} = \{v_t\} \uplus in(v_t).$$

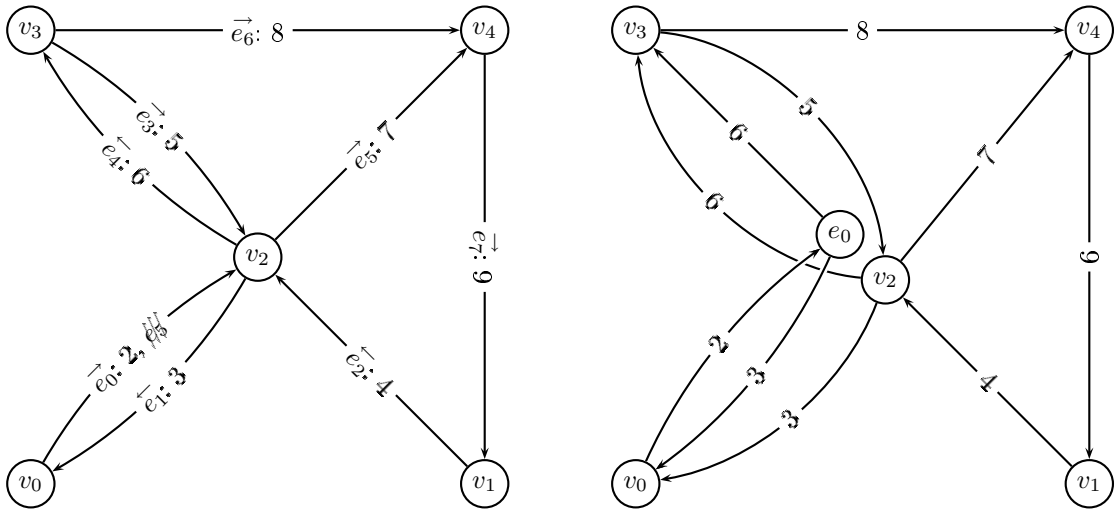
Damit können können sowohl M_s und M_t explizit angeben, als auch effizient ein Element von M auf Enthaltensein in diesen Mengen testen.

Das Ergebnis dieser Transformation ist der Algorithmus von Dijkstra zur Lösung des MSTSP auf einem Arbeitsgraphen mit M als Obermenge der Knotenmenge. Die Abbiegeverbote werden nicht mehr durch den Algorithmus berücksichtigt, sie sind vielmehr in den Arbeitsgraphen eingepflegt. Wir können also für diesen Arbeitsgraphen auch jeden anderen Algorithmus zur Lösung des MSTSP verwenden.

4.5.2 Beschreibung

Nachdem wir im letzten Unterabschnitt das Verfahren hergeleitet haben, fassen wir hier noch einmal die Konstruktion des Arbeitsgraphen zusammen. Wir transformieren einen gegebenen gewichteten Graphen $G(V, E, c)$ wie folgt in einen Arbeitsgraphen G_M:

- $G_M(V_M, E_M, c_M)$: Arbeitsgraph mit Knotenmenge V_M, Kantenmenge E_M und Kostenfunktion $c_M : E_M \to \mathbb{R}_+$.



Ausgangsgraph $G(V, E, c)$ Arbeitsgraph $G_M(V_M, E_M, c_M)$

Abbildung 3: Beispiel für die Transformation eines Graphen beim Knotensplitting

- Knotenmenge $V_M = V \uplus \sigma_{forbidden(e)\neq\emptyset}(E) \subseteq M$: Wir verwenden jetzt die Domäne $M = V \uplus E$ aus der Knoten- und Kantenaufahme als Obermenge der Knotenmenge für unseren Arbeitsgraphen. Dabei nehmen wir nur die $m \in M$ als Knoten in unseren Arbeitsgraphen auf, die beim Algorithmus der Knoten- und Kantenaufnahme überhaupt betrachtet werden können: Das sind genau alle Knoten und ferner alle Kanten, von denen ein Abbiegeverbot ausgeht.
- Kantenmenge $E_M = \bigcup out_M(V_M)$: Die Kantenmenge enthält für alle Knoten m alle Elemente in $out_M(m)$. Wir benötigen genau diese Kanten, da bei der Ausführung der Knoten- und Kantenaufnahme ausgehend von m_{min} genau über die Elemente aus $out_M(m_{min})$ iteriert wird.
- Kostenfunktion $c_M((m_1, m_2)) := c(k(m_1), k(m_2))$.

- M_s, M_t: Start- und Zielknotenmengen im Arbeitsgraphen. Wie oben beschrieben wählen wir $M_s := \{v_s\}$ und $M_t := (\{v_t\} \uplus in(v_t)) \cap V_M$. Wir beschränken M_t auf die Knoten, die tatsächlich in V_M enthalten sind. Die Menge M_t ist nicht leer, da zumindest $v_t \in V_M$ ist.

In Abbildung 3 ist ein Beispiel für eine derartige Transformation dargestellt. Wir haben den Ausgangsgraphen aus dem Beispiel bei der Kantenaufnahme in Abschnitt 4.2 verwendet. Im transformierten Graphen sind die Kanten nur mit ihren Kosten bezeichnet.

Von e_0 geht ein Abbiegeverbot aus, das über v_2 führt. Entsprechend ist e_0 auch als Knoten in unserem Arbeitsgraphen vertreten, von dem aus alle Knoten außer v_4 direkt erreicht werden können. Die im Ausgangsgraphen vorhandene Kante (v_0, v_2) ist im Arbeitsgraphen als Kante (v_0, e_0) vertreten.

Anschaulich gesprochen sind die Knoten v_2 und e_0 aus demselben Knoten v_2 im Ausgangsgraphen entstanden, v_2 wurde also „gesplittet“. (Es gilt $k(v_2) = k(e_0)$.) Der Knoten e_0 befindet sich im Arbeitsgraphen „auf einer höheren Ebene über v_2“, wobei sich v_2 (wie alle anderen Knoten aus $V \subseteq M$) „auf der untersten Ebene“ befindet. Jedesmal wenn die Traversierung des Arbeitsgraphen eine obere Ebene erreicht, bedeutet das, dass die entsprechende Vorgängerkante im Ausgangsgraphen ein Abbiegeverbot definiert. Die Ebene ist quasi das „Gedächtnis“ bei dem ansonsten

gedächtnislosen Algorithmus von DIJKSTRA.

Die Dissertation von SCHMID [Sch00] enthält einige weitere Beispiele.

4.5.3 Korrektheit

Das Verfahren ist eine Umwendlung der Knoten- und Kantenaufnahme in ein statisches Verfahren. Die Korrektheit folgt also bereits aus der Korrektheit der Knoten- und Kantenaufnahme und der Umformung. Im Folgenden führen wir zusätzlich einen direkten Korrektheitsbeweis.

Wir können zu jedem Pfad $P_M = \langle (m_0, m_1), (m_1, m_2), \ldots, (m_{n-1}, m_n) \rangle$ in G_M einen entsprechenden Pfad $P = \langle e_1, e_2, \ldots, e_n \rangle$ in G ohne Abbiegeverbote angeben: Wir wählen $e_i := (k(m_{i-1}), k(m_i))$. Ist zudem $m_0 \in M_s$ und $m_n \in M_t$, so folgt daraus sofort $\mathit{source}(e_1) = k(m_0) = v_s$ und $\mathit{target}(e_n) = k(m_n) = v_t$ aus der Definition von M_s bzw. M_t.

Es ist zunächst zu zeigen, dass $e_i \in E$ gilt. Dazu betrachten wir die entsprechende Kante im Arbeitsgraphen:

	$(m_{i-1}, m_i) \in E_M$	Voraussetzung	
$\Rightarrow$	$(m_{i-1}, m_i) \in \mathit{out}_M(m_{i-1})$	Definition von E_M	
$\Rightarrow$	$\exists e \in \mathit{out}(k(m_{i-1})) : m_i = h(e)$	Definition von out_M	(‡)
$\Rightarrow$	$\exists e \in \mathit{out}(k(m_{i-1})) : k(m_i) = k(h(e))$	Anwendung von k	
$\Rightarrow$	$\exists e \in \mathit{out}(k(m_{i-1})) : k(m_i) = \mathit{target}(e)$	$k \circ h = \mathit{target}$	
$\Rightarrow$	$\exists e \in E : \begin{array}{r} \mathit{source}(e) = k(m_{i-1}) \\ \wedge\, \mathit{target}(e) = k(m_i) \end{array}$	$\forall e \in \mathit{out}(v) : v = \mathit{source}(e)$	
$\Rightarrow$	$\exists e \in E : e = e_i$	Definition von e_i, G hat keine Multikanten	
$\Rightarrow$	$e_i \in E$		

Der Pfad P ist offensichtlich gültig, da die Gleichung

$$\mathit{target}(e_i) = k(m_i) = k(m_{(i+1)-1}) = \mathit{source}(e_{i+1})$$

gilt.

Es bleibt zu zeigen, dass P kein Abbiegeverbot enthält. Dazu zeigen wir zunächst die stärkere Aussage $m_i = h(e_i)$. Der Beweis dafür sieht analog zum vorhergehenden aus, mit dem Unterschied, dass wir in Zeile (‡) zusätzlich $e \neq e_i$ annehmen und diese Annahme schlussendlich wegen des Verbots von Multikanten zum Widerspruch führen.

Angenommen, P enthielte ein Abbiegeverbot. Sei $i \in \{1, \ldots, n-1\}$ der kleinste Index, so dass $(e_i, e_{i+1}) \in T$ ist. Wir zeigen zunächst $m_i = e_i$:

	$\mathit{forbidden}(e_i) \neq \emptyset$	$(e_i, e_{i+1}) \in T$
$\Rightarrow$	$h(e_i) = e_i$	Definition von h
$\Rightarrow$	$m_i = e_i$	$m_i = h(e_i)$

Im Folgenden führen wir die Voraussetzung für i zum Widerspruch:

	$(m_i, m_{i+1}) \in E_M$	Voraussetzung
$\Rightarrow$	$(e_i, m_{i+1}) \in E_M$	$m_i = e_i$
$\Rightarrow$	$(e_i, m_{i+1}) \in \mathit{out}_M(e_i)$	Definition von E_M
$\Rightarrow$	$(e_i, m_{i+1}) \in \{(e_i, h(e)) : e \in \mathit{out}(k(e_i)) \setminus \mathit{forbidden}(e_i)\}$	Definition von out_M

$\Rightarrow$	$m_{i+1} \in \{h(e) : e \in \mathit{out}(k(e_i)) \setminus \mathit{forbidden}(e_i)\}$	Projektion
$\Rightarrow$	$m_{i+1} \in \{h(e) : e \in \mathit{out}(\mathit{target}(e_i)) \setminus \mathit{forbidden}(e_i)\}$	$e_i \in E$, Definition von k
$\Rightarrow$	$k(m_{i+1}) \in k(\{h(e) : e \in \mathit{out}(\mathit{target}(e_i)) \setminus \mathit{forbidden}(e_i)\})$	Anwendung von k
$\Rightarrow$	$k(m_{i+1}) \in \{\mathit{target}(e) : e \in \mathit{out}(\mathit{target}(e_i)) \setminus \mathit{forbidden}(e_i)\}$	$k \circ h = \mathit{target}$
$\Rightarrow$	$k(m_{i+1}) \in \begin{array}{l}\{\mathit{target}(e) : e \in \mathit{out}(\mathit{target}(e_i))\} \\ \setminus \{\mathit{target}(e) : e \in \mathit{forbidden}(e_i)\}\end{array}$	G hat keine Multikanten
$\Rightarrow$	$k(m_{i+1}) \notin \{\mathit{target}(e) : e \in \mathit{forbidden}(e_i)\}$	Mengenlehre
$\Rightarrow$	$\mathit{target}(e_{i+1}) \notin \{\mathit{target}(e) : e \in \mathit{forbidden}(e_i)\}$	Definition von e_i
$\Rightarrow$	$e_{i+1} \notin \mathit{forbidden}(e_i)$	G hat keine Multikanten
$\Rightarrow$	Widerspruch zu $(e_i, e_{i+1}) \in T$	

Wir haben unsere Wahl von i zum Widerspruch geführt. Folglich enthält P keine Abbiegeverbote.

Für die Pfadkosten gilt $c(P) = c_M(P_M)$ nach Konstruktion und Definition von c_M.

Auf analoge Art und Weise definieren wir zu einem gegebenen gültigen abbiegeverbotfreien Pfad $P = \langle e_1, e_2, \ldots, e_n \rangle$ in G einen entsprechenden Pfad $P_M = \langle (m_0, m_1), (m_1, m_2), \ldots, (m_{n-1}, m_n) \rangle$ in G_M: Wir setzen $m_0 := \mathit{source}(e_1)$ und $m_i := h(e_i)$. Für $\mathit{source}(e_1) = v_s$ und $\mathit{target}(e_n) = v_t$ gilt ferner $m_0 = v_s \in M_s$ und $m_n = h(e_n) \in \{v_t, e_n\} \cap V_M$, also $m_n \in M_t$.

Zu zeigen bleibt, dass alle Knotenpaare in P_M Kanten in G_M sind, also dass stets $(m_{i-1}, m_i) \in E_M$ gilt. Dazu zeigen wir zunächst $k(m_{i-1}) = \mathit{source}(e_i)$ für $i \in \{1, \ldots, n\}$:

	$m_i = h(e_i)$	Definition P_M
$\Rightarrow$	$k(m_i) = k(h(e_i))$	Anwendung von k
$\Rightarrow$	$k(m_i) = \mathit{target}(e_i)$	$k \circ h = \mathit{target}$
$\Rightarrow$	$i < n \wedge k(m_i) = \mathit{source}(e_{i+1})$	Gültigkeit von P
$\Rightarrow$	$i > 1 \wedge k(m_{i-1}) = \mathit{source}(e_i)$	Indexverschiebung, $1 \leq i \leq n$
$\Rightarrow$	$\begin{array}{r}(i > 1 \wedge k(m_{i-1}) = \mathit{source}(e_i)) \\ \vee (i = 1 \wedge k(m_{i-1}) = \mathit{source}(e_i))\end{array}$	Definition P_M: $m_0 = \mathit{source}(e_1)$
$\Rightarrow$	$k(m_{i-1}) = \mathit{source}(e_i)$	$1 \leq i$

Nach Definition von E_M ist $(m_{i-1}, m_i) \in \mathit{out}_M(m_{i-1})$ für $i \in \{1, \ldots, n\}$ zu zeigen. Wir suchen also ein $e \in \mathit{out}(k(m_{i-1}))$, so dass $h(e) = m_i$ ist. Für $m_{i-1} \in E$ muss ferner $e \notin \mathit{forbidden}(m_{i-1})$ gelten. Der Beweis ist vollbracht, wenn wir immer ein solches e angeben können.

Wir zeigen, dass $e := e_i$ die oben genannten Forderungen erfüllt. Der Sachverhalt $e_i \in \mathit{out}(k(m_{i-1}))$ folgt sofort aus der zuletzt bewiesenen Gleichung. Ferner bemerken wir, dass $h(e_i) = m_i$ nach Definition von P_M gilt. Es bleibt noch $e_i \notin \mathit{forbidden}(m_{i-1})$ für $m_{i-1} \in E$ zu zeigen:

	$m_{i-1} \in E$	Voraussetzung
$\Rightarrow$	$i > 1$	$m_0 \in V$
$\Rightarrow$	$m_{i-1} = h(e_{i-1})$	Definition von P_M
$\Rightarrow$	$m_{i-1} = e_{i-1}$	Definition von h
$\Rightarrow$	$\mathit{forbidden}(m_{i-1}) = \mathit{forbidden}(e_{i-1})$	Anwendung von *forbidden*
$\Rightarrow$	$e_i \notin \mathit{forbidden}(m_{i-1})$	$e_i \notin \mathit{forbidden}(e_{i-1})$

Also enthält P_M in der Tat nur Kanten aus E_M.

Der Pfad P_M ist trivialerweise gültig.

Wir haben jedem Pfad P_M in G_M eindeutig einen abbiegeverbotfreien Pfad P in G zugeordnet, und umgekehrt. Insbesondere ist unsere Zuordnung eine Bijektion: Man vergleiche die Konstruktion der Zuordnung von P auf P_M mit der bewiesenen Eigenschaft $m_i = h(e_i)$ und der offensichtlichen Eigenschaft $m_0 = source(e_1)$ bei der Zuordnung von P_M auf P. Dadurch bleiben auch bei der Zuordnung von G nach G_M die Pfadkosten gleich, da die Gleichheit der Kosten für die Zuordnung von G_M nach G bereits gezeigt wurde.

Die Zuordnung behält die Relation zwischen v_s und M_s bzw. v_t und M_t bei. Das bedeutet, dass ein kürzester Pfad in G_M zwischen einem Knoten aus M_s und einem Knoten aus M_t einem kürzesten Pfad ohne Abbiegeverbote in G zwischen v_s und v_t entspricht. Wir können also jeden Algorithmus, der das MSTSP löst, auf unseren Arbeitsgraphen G_M ansetzen, um einen abbiegeverbotfreien Pfad im Ausgangsgraphen G zu erhalten. □

Wenn wir in unserem Beispiel in Abbildung 3 wieder den kürzesten Weg von v_0 nach v_4 unter Berücksichtigung der Abbiegeverbote suchen, erhalten wir $M_s = \{v_0\}$ und $M_t = \{v_4\}$, letzteres wegen $\sigma_{forbidden(e)\neq\emptyset}(in(v_t)) = \emptyset$. Im Arbeitsgraphen kann v_0 nur zum Knoten e_0 hin verlassen werden, der wiederum v_3 als einzigen Nachbarn abgesehen von v_0 hat. Der kürzeste Weg von v_3 zu v_4 ist der direkte Weg. Der Weg $P_M = \langle(v_0, e_0), (e_0, v_3), (v_3, v_4)\rangle$ im Arbeitsgraphen entspricht $P = \langle(v_0, v_2), (v_2, v_3), (v_3, v_4)\rangle = \langle e_0, e_4, e_6\rangle$ im Ausgangsgraphen. P ist der kürzeste abbiegeverbotfreie Weg zwischen v_0 und v_4 in G.

4.5.4 Minimierung des Arbeitsgraphen

Der so erzeugte Arbeitsgraph G_M ist nicht minimal: Er enthält Knoten und Kanten, die entfernt werden könnten, ohne die Transformation zu beeinträchtigen. In [Sch00] wird eine ähnliche Graph-Transformation angegeben, die diese Minimalitätseigenschaft besitzt. Wir beschränken uns hier darauf, die Unterschiede aufzuzeigen, den Ursprung dieser Unterschiede zu begründen und die Verwendung eines nicht minimalen Arbeitsgraphen zu rechtfertigen.

Dazu definieren wir wieder eine Äquivalenzrelation $\sim$, diesmal für Knoten. Wir bezeichnen zwei Knoten als äquivalent, wenn die Ausgangskanten zu jeweils denselben Knoten führen und die Kosten jeweils gleich sind. Auf eine formale Darstellung verzichten wir hier.

Äquivalente Knoten können entstehen, wenn im Ausgangsgraphen zwei Kanten e_1 und e_2 mit demselben Zielknoten dieselbe nichtleere Abbiegeverbotsmenge haben, also wenn

$$target(e_1) = target(e_2) \wedge forbidden(e_1) = forbidden(e_2) \neq \emptyset$$

gilt. Wir geben in Abbildung 4 ein Beispiel an, das im Vergleich zum vorherigen leicht modifiziert ist: Jetzt geht auch von Kante e_2 ein Abbiegeverbot in Richtung e_5 aus. Die Knoten e_0 und e_2 im Arbeitsgraphen sind äquivalent bezüglich $\sim$, da die Ausgangskantenmenge jeweils aus einer Kante zu v_3 mit Kosten 6 besteht.

Sind zwei Knoten $v_1 \sim v_2$ äquivalent, ist es dem Algorithmus von DIJKSTRA „gleichgültig“, welcher Knoten aus der Prioritätswarteschlange entnommen wird: Die Schritte in der inneren Schleife sind für beide Knoten dieselben. Wir können also einen Graphen mit zwei äquivalenten Knoten v_1 und v_2 minimieren, wenn wir die Eingangskanten von v_2 auf v_1 zeigen lassen und v_2 und alle Ausgangskanten von v_2 entfernen. In unserem Beispiel würde dann z.B. die Kante mit Kosten 4 von v_1 aus auf e_0 zeigen und e_2 mit den zugehörigen Ausgangskanten entfernt werden.

Bei unsererer Transformation würden keine voneinander verschiedene äquivalente Knoten entstehen, wenn wir die Äquivalenzrelation $\equiv$ aus Abschnitt 4.4.1 feiner gewählt hätten. Die Äquivalenzrelation $\doteq$ sieht Kanten nur dann als äquivalent an, wenn sie den gleichen Zielknoten haben und die Verbotsmengen gleich sind:

$$e_1 \doteq e_2 \iff target(e_1) = target(e_2) \wedge forbidden(e_1) = forbidden(e_2).$$

(Unsere bisherige Definition erfasst nur Kanten mit leerer Verbotsmenge als äquivalent.) Die Äquivalenzrelation $\doteq$ ist eine Verfeinerung von $\equiv$, da aus $e_1 \equiv e_2$ auch $e_1 \doteq e_2$ folgt. Daher hätten wir bei der Herleitung der Knoten- und Kantenaufnahme auch $\doteq$ verwenden können, ohne den

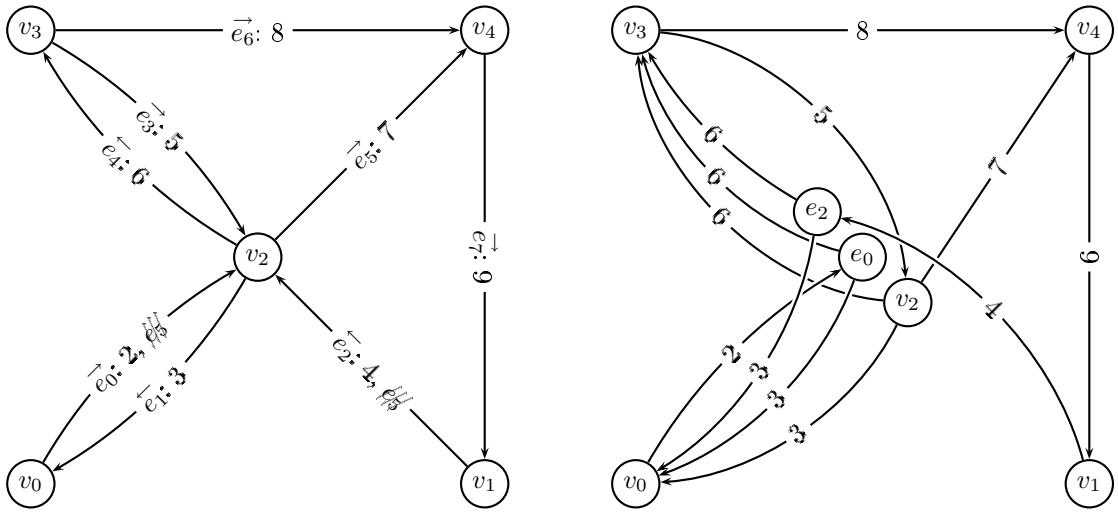


Abbildung 4: Beispiel für äquivalente Knoten bei der Transformation eines Graphen beim Knotensplitting

Rest der Herleitung zu verändern. Man kann zeigen, dass dann der Arbeitsgraph keine nichttrivial äquivalenten Knoten hat, wenn der Ausgangsgraph ebenfalls keine nichttrivial äquivalenten Knoten besitzt.

Wir favorisieren jedoch die vorgestellte Variante der Äquivalenzrelation $\equiv$, selbst wenn wir dadurch Gefahr laufen, überflüssige Knoten und Kanten im Arbeitsgraphen zu bekommen. Zum einen stellen wir fest, dass nur bei unserer Definition der Äquivalenzrelation die Funktion h wirklich schnell on-line berechnet werden kann. Für $\doteq$ müssen wir zumindest die Abbiegeverbotsmenge vorher genauer untersuchen und eine redundante Datenstruktur (z.B. ein Array von Kanten auf Äquivalenzklassen) erzeugen, die die Abbildung $h_{\doteq}$ in $O(1)$ realisiert. Das kostet im schlimmsten Fall $O(|E|)$ zusätzlichen Speicher. Zum anderen haben wir bei den uns zur Verfügung stehenden realen Straßengraphen einiger Länder Europas nur bei einem Land einen Anteil redundanter Knoten von mehr als einem Prozent festgestellt. In allen anderen Ländern lag der Anteil unter 7 ‰. (Siehe auch Abschnitt 6.1.) Das führt uns zur Vermutung, dass unsere Definition der Äquivalenz für den gegebenen Anwendungsfall „hinreichend fein“ ist und eine weitere Verfeinerung kaum etwas zur Verbesserung der Laufzeit beitragen würde.

4.5.5 Implementation

Wie bereits bei der Kantenaufnahme betrachten wir jetzt die effiziente Realisierbarkeit der DIJKSTRA-Zugriffsoperationen aus Unterabschnitt 2.2.3. Wir betrachten dabei die Funktionen out_M und c_M näher, da die beiden anderen Operationen Gleichheitstest und Start- bzw. Zielknoten offensichtlich effizient implementiert werden können.

Die Funktion out_M verwendet die Funktionen h, k, *forbidden* und *out*. Die ersten beiden können in $O(1)$ berechnet werden, die dritte benötigt bei einer geeigneten Implementation als Array von kleine Bitvektoren auch nur konstante Laufzeit. Die Funktion *out* benötigt bei der Implementation des Graphen als Adjazenzliste für die Berechnung von $out(v)$ Zeit in der Ordnung der Größe der Ausgangskantenmenge $outdeg(v)$. Streng genommen erfüllt out_M unsere eingangs im Unterabschnitt 2.2.3 gestellte Forderung für die Laufzeit nicht: Für $m \in E$ gilt die Ungleichung

$outdeg_M(m) < outdeg(k(m))$, demnach werden mehr Elemente betrachtet als zurückgegeben. Für Straßengraphen spielt dieser minimale Unterschied wegen beschränktem Ausgangsgrad jedoch kaum eine Rolle.

Für die Berechnung der Funktion c_M bemerken wir, dass hierfür die Funktionen k und c benötigt werden. Die Funktion k benötigt $O(1)$ Laufzeit, also kann c_M genauso effizient wie c berechnet werden. Bei einer naheliegenden Implementation als Array von Fließkommazahlen beträgt auch hier die Zugriffszeit $O(1)$.

Allerdings liegt die Berechnung von in_M, wie sie z.B. für die bidirektionale Suche notwendig ist (siehe 2.3.3), nicht auf der Hand. Zur Berechnung von $in_M(m)$ ist eine Untersuchung der Vorgänger der Kanten aus $in(k(m))$ (also der „Vorvorgänger" von $k(m)$) notwendig. Das Problem wurde im Rahmen der Referenz-Implementation betrachtet und gelöst, eine tiefer gehende Betrachtung würde hier jedoch den Rahmen sprengen. Diese Berechnung ließe sich durch eine geeignete Vorverarbeitung der Abbiegeverbots-Menge beschleunigen.

4.5.6 Komplexität

Die Laufzeit hängt wie bei der Knotenaufnahme in erster Linie von der Größe des Graphen ab. Hier gilt

$$|V_M| = |V \uplus \{e : forbidden(e) \neq \emptyset\}| < |V| + \min(|T|, |E|).$$

Für die von uns untersuchten Straßengraphen gilt stets $|T| < |V| < |E|$, hier können wir die Knotenanzahl mit

$$|V_M| < |V| + |T| < 2 \cdot |V|$$

nach oben abschätzen.

Für der Kantenmenge zeigen wir, dass sie unter der Voraussetzung beschränkten Eingangs- und Ausgangsgrads deg die Ungleichung

$$|E_M| < (|V| + \min(|T|, |E|)) \cdot deg$$

gilt:

$$\begin{array}{rcll} |E_M| & = & \sum_{m \in V_M} |out_M(m)| & \text{Definition } E_M \\ & \leq & \sum_{v \in V} |out_M(v)| + \sum_{\substack{e \in E: \\ forbidden(e) \neq \emptyset}} |out_M(e)| & \text{Definition von } V_M \\ & \leq & |V| \cdot deg + \min(|T|, |E|) \cdot deg & \text{Definitionen von } out_M \text{ und } forbidden \\ & = & (|V| + \min(|T|, |E|)) \cdot deg & \end{array}$$

Für den schlimmsten Fall können wir $|E_M|$ mit

$$|E_M| < 2 \cdot |E| \cdot deg$$

abschätzen. Für $|T| < |V| < |E|$ und mit $|E| < |V| \cdot deg$ erhalten wir jedoch

$$|E_M| < 2 \cdot |E|.$$

Die Größe der Knoten- und Kantenmengen des Arbeitsgraphen unterscheidet sich bei dünnen Graphen auch im schlimmsten Fall nur um einen konstanten Faktor von der Größe der Knoten- und Kantenmengen im Ausgangsgraphen. Die asymptotische Laufzeit beim Knotensplitting entspricht also der des Algorithmus von DIJKSTRA: $O(n \log n)$. Bei in der Praxis auftretenden Fällen wird der Faktor für den Größenunterschied nur knapp größer als eins sein, daher können wir von einer nur leicht erhöhten Laufzeit ausgehen. Wir werden das in Abschnitt 6.2 empirisch überprüfen.

4.5.7 Bewertung

Im Gegensatz zur Kantenaufnahme erzeugt dieses Verfahren einen wesentlich kleineren Arbeitsgraphen und verspricht somit eine im Vergleich wesentlich bessere Laufzeit. Es kann auch bequem on-line angewendet werden. Obwohl der erzeugte Arbeitsgraph nicht minimal ist, lohnt sich die Minimierung für unseren Anwendungsfall nicht, da der erwartete Laufzeitvorteil gering ist und dadurch die Implementation eines on-line-Verfahrens erschwert wird.

4.5.8 Allgemeine Wegeverbote

Wir haben uns bisher nur mit Abbiegeverboten, also Wegeverboten der Länge 2, auseinander gesetzt. Die Erweiterung des Knotensplittings auf allgemeine Wegeverbote soll hier nur kurz skizziert werden, eine ausführliche Beschreibung mit Beweis kann in [Sch00] nachgeschlagen werden.

Das Knotensplitting basiert auf folgender Idee: Die Information, ob im nächsten Schritt ein Abbiegeverbot zu berücksichtigen ist, wird nicht im Algorithmus, sondern im Zustand der Suche verwaltet. Befindet sich die Suche auf einem Knoten einer „höheren Ebene“, bedeutet das, dass im nächsten Schritt aufgrund eines Abbiegeverbots nicht alle Knoten im Ausgangsgraphen erreicht werden können – was durch das Fehlen der entsprechenden Kanten im Arbeisgraphen realisiert wird.

Dieselbe Idee kann auch für allgemeine Wegeverbote verfolgt werden. Wenn während der Suche ein Wegeverbot betreten wird, wird die Suche im Arbeitsgraph auf eine bestimmte, nur diesem Wegeverbot zugeordnete Ebene geführt. Wenn das Wegeverbot vor der letzten Verbotskante verlassen wird, wird die Suche wieder auf die „Standard-Ebene“ zurückgeführt, die keinem Wegeverbot entspricht. Das Wegeverbot kann aber nicht über die letzte Verbotskante verlassen werden, da die entsprechende Kante für diese Ebene im Arbeitsgraphen fehlt.

Da Wegeverbote sich überlappen können, scheint für eine effiziente Zuordnung der „Ebenen“ im Arbeitsgraphen zu den aktiven Wegeverboten zumindest eine Vorberechnung auf der Wegeverbotsmenge notwendig zu sein. Nach einer derartigen Vorberechnung wäre ein on-line-Verfahren ähnlich zum Knotensplitting denkbar. Allerdings wäre ein on-line-Verfahren für allgemeine Wegeverbote vermutlich um einiges komplexer als das Knotensplitting für Abbiegeverbote.

Für reale Anwendungen werden die meisten Wegeverbote die Länge 2 haben. Ein Hybrid-Ansatz, bei dem „problematische“ Wegeverbote mit mehr als zwei Kanten vorab off-line und die Abbiegeverbote erst on-line transformiert werden, scheint hier ein guter Ansatz zu sein.

5 Referenz-Implementation

Die hier vorgestellten Verfahren wurden implementiert. Um die Algorithmen zu testen, wurde eine kleine Beispiel-Anwendung mit Kommandozeilen-Schnittstelle entwickelt. In diesem Abschnitt möchten wir unsere Beispiel-Anwendung und ihre Möglichkeiten und Grenzen vorstellen. Wir gehen etwas genauer auf die Implementation der Algorithmen ein. Dabei stellen wir auch den Entwurf vor, der für eine reale Anwendung adaptiert werden kann. Die Probleme, die dabei aufgetreten sind, und die Lösungen dafür sollen hier kurz als Fallstudie skizziert werden. Das kann helfen, einen Überblick über die zu erwartenden Probleme bei einer realen Anwendung zu erhalten.

5.1 Beispiel-Anwendung zum Test der Algorithmen

Unsere Beispiel-Anwendung ist kommandozeilengesteuert und leistet unter anderem Folgendes:

- Einlesen eines Graphen aus einer Datei oder von der Standard-Eingabe.
- Unterstützung verschiedener Such-Algorithmen, einfache Erweiterbarkeit.
- Ausführung eines oder mehrerer Algorithmen für gegebene oder zufällig gewählte Start- oder Zielknoten mit einer gegebenen Anzahl Iterationen.

- Ausgabe des kürzesten Pfads (falls vorhanden) und dessen Länge, Prüfung des Pfads auf Gültigkeit und Enthalten von Abbiegeverboten.
- Messung und Ausgabe der verbrauchten CPU-Zeit.
- Erfassung und Ausgabe statistischer Daten.
- Wahlweise Ausgabe in Dateien oder auf die Standardausgabe.
- Initialisieren des verwendeten Pseudo-Zufallsgenerators mit einem vorgegebenen Startwert.
- Verschiedene nicht problembezogene Tests wie z.B. Bestimmung der starken Zusammenhangskomponenten.

Es waren bereits Graphen vorhanden, die das Straßennetz jeweils eines europäischen Landes repräsentierten. Die Graphen lagen im GraphML-Format [Gra] vor, Routinen zum Einlesen gab es auch schon. Allerdings waren die Ausgangsdaten nicht für eine direkte Verarbeitung geeignet: Der Graph lag als ungerichteter Graph vor, es gab Multikanten, und es gab Abbiegeverbote, die nicht zugeordnet werden konnten.

Um die Säuberung der Daten von der eigentlichen Anwendung zu trennen, wurde ein separates Programm entwickelt, das die Graphen aus dem GraphML-Format eingelesen, „gesäubert“ und in das eigens entwickelte GPR-Format konvertiert hat. Das GPR-Format (Graph with Path Restrictions) vereint den Graphen mit seinen Abbiegeverboten in einer Datei. Eine Grammatik in EBNF-Form und eine Beispiel-Datei sind in Anhang C zu finden. Für das Einlesen der GPR-Dateien wurde mit Hilfe von flex und bison [Her92] ein schneller tabellengesteuerter Parser generiert.

Dadurch konnte die Beispiel-Anwendung schlank gehalten werden, da sie von sauberen, korrekten Eingabedaten ausgehen konnte. Ferner konnte das Einlesen um ein Vielfaches beschleunigt werden. Das Format ist sehr gut geeignet, um kleine Beispiele zu erzeugen. Es ist kompakt, da sowohl der Graph als auch die Abbiegeverbote in einer Datei Platz finden. Außerdem kann der Parser leicht an Erweiterungen des Formats angepasst werden, z.B. für die Einbeziehung allgemeiner Wegeverbote.

5.2 Implementation der Algorithmen

Für die Entwicklung der Algorithmen stand die Bibliothek LEDA (Library of Efficient Data types and Algorithms) [NM99] zur Verfügung. LEDA bietet für die Programmiersprache C++ unter anderem effiziente Datenstrukturen für Graphen und für Wörterbücher. Dadurch konnte bei der Implementation der Fokus auf die problembezogenen Aspekte gesetzt werden.

LEDA arbeitet verstärkt mit C++-Templates [Wil04]. Dieses Paradigma haben wir, wo es sinnvoll war, auch für unser Projekt verwendet. Vorteile sind z.B. Typsicherheit und eine bessere Optimierbarkeit des Codes als bei virtuellen Funktionen.

Nach der Entwicklung der ersten rudimentären Version der Kommandozeilenschnittstelle und der Routine zum Einlesen der Graphen wurde als erstes die bei LEDA mitgelieferte Implementation des Algorithmus von Dijkstra verwendet. Diese kennt zwar keine Abbiegeverbote, ist aber gut als Vergleich für die später hinzugefügten Verfahren geeignet. Für die dynamischen Algorithmen „Mehrfach-Knotenaufnahme“ und „Knoten- und Kantenaufnahme“ wurde die mitgelieferte Implementation entsprechend der Definition des Algorithmus angepasst.

Um Erfahrungswerte für die Integration von statischen Verfahren in bestehende Umgebungen zu gewinnen und die „on-line-Fähigkeit“ unserer Algorithmen zu verifizieren, musste in unserem Fall zunächst eine solche Umgebung geschaffen werden. Dazu wurde zunächst eine beschleunigte Implementation des Dijkstra-Algorithmus in der bigerichteten Variante (siehe Abschnitt 2.3) abgewandelt. Diese Implementation verwendete die durch LEDA definierten Zugriffsoperationen auf den Graphen. Die LEDA-Zugriffsoperationen, die den unterstrichenen Operationen im Pseudocode in Anhang A auf Seite 40 entsprechen, wurden durch Zugriffsoperationen auf eine eigene

(bislang abstrakte) Graph-Datenstruktur ersetzt. Im nächsten Schritt wurde der Algorithmus für das MSTSP erweitert. Das waren die einzigen Änderungen am Algorithmus von DIJKSTRA.

Nun galt es, Implementationen für die eigene Graph-Datenstruktur zu entwickeln, die on-line für den MSTSP-Algorithmus ausgehend von einem Graphen mit Abbiegeverboten eine Sicht auf einen Arbeitsgraphen ohne Abbiegeverbote präsentieren. Wir bezeichnen diese Implementationen als Adapter. Der erste Schritt lief auch hier auf das Weglassen von Abbiegeverboten hinaus – es wurde ein „Identitäts-Adapter“ entwickelt, der einfach einen LEDA-Graphen 1:1 in die eigene Graph-Datenstruktur umgewandelt hat. Die beiden anderen Adapter für die „Kantenaufnahme“ und für das „Knotensplitting“ folgten und konnten aufgrund des vorher durchgeführten Entwurfs nahtlos integriert werden.

Um einen Vergleich zwischen on-line- und off-line-Verfahren herstellen zu können, wurde zu jedem on-line-Verfahren das zugehörige off-line-Verfahren entwickelt. Dafür war lediglich die Implementation eines weiteren Adapters, des „generischen off-line-Adapters“, notwendig. Dieser erhält als Template-Parameter eine andere Adapter-Klasse und benutzt deren Zugriffs-Operationen, um mittels einer Tiefensuche eine Repräsentation des Graphen im Arbeitsspeicher aufzubauen.

Insgesamt können wir festhalten, dass die Entwicklungszeit durch die Benutzung der LEDA-Klassenbibliothek drastisch verkürzt wurde, obwohl vorher eine Einarbeitung notwendig war. Frühzeitig durchgeführte Überlegungen zum Entwurf haben ebenfalls Zeit bei der Implementation gespart. Die Integration eines statischen Verfahrens in einen bestehenden Algorithmus, der eine Beschleunigungstechnik anwendet, kann recht einfach durchgeführt werden, indem anstelle eines Graphen im Arbeitsspeicher auf die entsprechende Adapter-Klasse aus unserer Referenz-Implementation zugegriffen wird. Allerdings lassen unsere Implementationen noch sehr großzügigen Spielraum für Optimierungen, nicht zuletzt fordert die Bequemlichkeit einer Klassenbibliothek ihren Preis bei der Laufzeit des so erzeugten Codes.

5.3 Bei der Implementation aufgetretene Probleme

Quelldaten Ein Problem waren die unsauberen Daten, für die zunächst eine Säuberungs-Routine geschrieben werden musste. Das kann auch in bereits bestehenden Anwendungen auftreten, wo die Abbiegeverbote bisher zwar vorhanden waren, aber nicht benutzt wurden. Allerdings ist dieses Problem eher technischer Natur.

Datenstruktur-Inkompatibilität Bei der Anpassung der LEDA-Variante der bigerichteten Suche fiel auf, dass die Prioritätswarteschlange mit *decrease_key*-Funktion zwar für Knoten, aber nicht für allgemeine Datentypen zur Verfügung stand. Da die Implementation der Kürzesten-Wege-Suche für Adapter möglichst generisch gehalten werden sollte, musste die Prioritätswarteschlange für unseren Algorithmus selbst implementiert werden. Dafür wurde eine kombinierte Datenstruktur aus Prioritätswarteschlange und Wörterbuch von Kanten auf Warteschlangen-Einträge entworfen. Wenn die Kürzeste-Wege-Suche nur für einen speziellen Adapter zu implementieren gewesen wäre, hätte die Wörterbuch-Struktur besser auf die vom Adapter verlangte Knotenmenge angepasst werden können. Als Fazit können wir festhalten, dass die von uns verfolgte Allgemeinheit und Unterstützung verschiedener Verfahren höchstens für akademische Zwecke geeignet ist: Bei einer praktisch orientierten Implementation wird man vorab ein Verfahren (z.B. das „Knotensplitting“) wählen, einen speziellen optimierten Adapter für eine speziell optimierte Schnittstelle entwickeln und die Implementation der Kürzesten-Wege-Suche auf diesen Adapter ausrichten.

MSTSP Ein weiteres Problem bei den statischen Adaptern war, dass der Zielknoten (und bei der Kantenaufnahme auch der Startknoten) nicht mehr eindeutig bestimmt waren. Das führte uns zur Definition des MSTSP. In der Praxis wird man dafür aus Performance-Gründen eine semi-statische Variante wählen, die das Erreichen des Zielknotens (bzw. der Zielkante, je nach Problemstellung) effizient prüft. Wir arbeiten bei unserer Referenz-Implementation mit einer explizit angegebenen Zielknotenmenge, was für die Laufzeit nicht gerade förderlich sein kann.

Eine Möglichkeit, das Problem zumindest beim Knotensplitting ganz zu umgehen, ist die Forderung, dass nach Erreichen des Ziels dieses über jede Kante verlassen werden kann. Dies ist aber

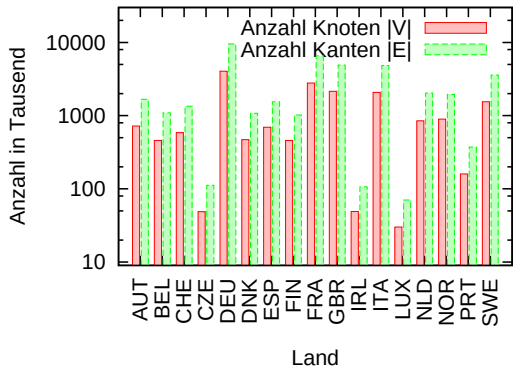


Abbildung 5: Anzahl Knoten und Kanten in den Testgraphen

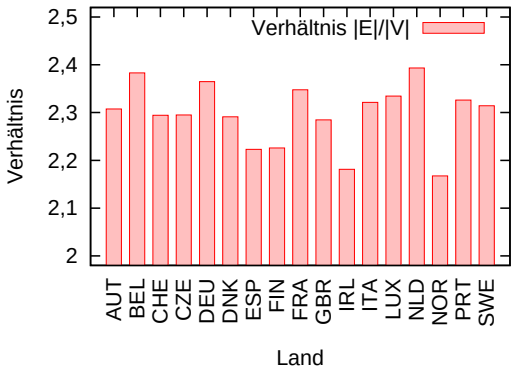


Abbildung 6: Verhältnis zwischen Kanten- und Knoten-Anzahl in den Testgraphen

nicht immer möglich, da es Knoten geben kann, bei denen jede Eingangskante Verbotseingang ist. Sicherlich wird man auch bei einer realen Implementation auf das Problem der mehreren Zielknoten stoßen und je nach verwendeter Beschleunigungstechnik und Problemdefinition eine Lösung finden. Wir weisen hier lediglich darauf hin, dass das Problem wahrscheinlich für alle Anwendungsfälle existiert; eine allgemeine Lösung für jeden dieser Anwendungsfälle anzugeben ist nicht möglich.

6 Empirische Analyse

Im folgenden Abschnitt werden wir die Ergebnisse der empirischen Analyse unserer Algorithmen vorstellen. Wir beginnen mit einer kurzen Vorstellung einiger statischer Eigenschaften der Graphen, auf denen unsere Algorithmen getestet wurden. Anschließend zeigen wir die Ergebnisse der Laufzeittests.

6.1 Testgraphen

6.1.1 Überblick

Bei unseren Tests haben wir reale Straßengraphen einiger Länder Europas verwendet. Die Anzahl Knoten reicht dabei von etwa 30.000 (Luxemburg) bis zu 4.000.000 (Deutschland), so dass wir Laufzeiten für Graphen sehr unterschiedlicher Größen angeben können. Die Kantenanzahl ist ein vergleichsweise konstantes Vielfaches der Knotenanzahl, das Verhältnis Kanten- zu Knotenanzahl liegt zwischen 2,1 und 2,4. Abbildung 5 zeigt eine Übersicht über die Anzahl Knoten und Kanten. In Abbildung 6 ist das Verhältnis zwischen Kanten- und Knotenanzahl dargestellt.

6.1.2 Relative Anzahl der Abbiegeverbote

Um einen Überblick über die relative Anzahl der Abbiegeverbote zu erhalten, haben wir für die Testgraphen die Anzahl Abbiegeverbote ins Verhältnis zur Knotenanzahl gesetzt. In den meisten Ländern liegt das Verhältnis um 5 %. Ausreißer sind Spanien und Portugal, dort beträgt das Verhältnis etwa 20 %. Ein entsprechendes Balkendiagramm ist in Abbildung 7 angegeben. Aus diesem Grund untersuchen wir die Abhängigkeit der Laufzeit von der Anzahl der Abbiegeverbote nicht: In der Praxis hat die Größe des Graphen einen weit stärkeren Einfluss auf die Laufzeit als die Anzahl der Abbiegeverbote.

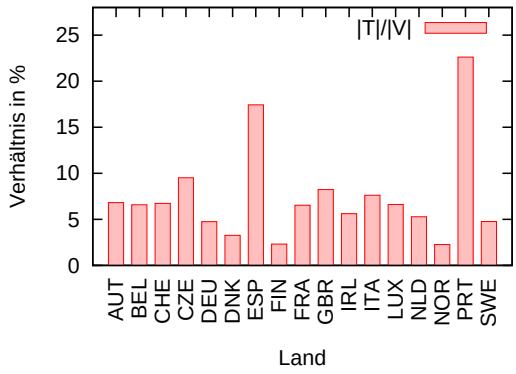


Abbildung 7: Verhältnis zwischen der Anzahl Abbiegeverbote und der Knotenanzahl in den Testgraphen

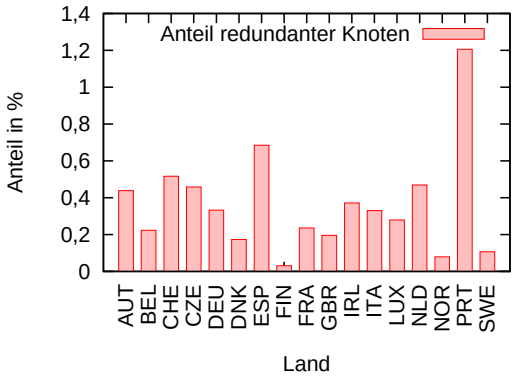


Abbildung 8: Anteil der beim Knotensplitting entstehenden redundanten Knoten in den Testgraphen

6.1.3 Beim Knotensplitting entstehende redundante Knoten

Auch die Anzahl der beim Knotensplitting entstehenden redundanten Knoten wurde für die einzelnen Graphen untersucht. Nur für Portugal sind mehr als ein Prozent der Knoten redundant, bei allen anderen Ländern liegt der Anteil redundanter Knoten zum Teil weit darunter. Die beiden Länder mit dem hohen Anteil an Abbiegeverboten erzeugen erwartungsgemäß auch die meisten redundanten Knoten. Abbildung 8 verdeutlicht diesen Sachverhalt.

6.2 Laufzeit

Wir haben eine Testserie mit allen Graphen, allen Algorithmen und jeweils 200 Suchläufen für jeweils dieselben Start- und Zielknoten durchgeführt. Die Start- und Zielknoten wurden pseudozufällig gewählt, die ermittelten Wege sind daher relativ lang. Da wir eine nicht beschleunigte Variante des Algorithmus von Dijkstra verwendet haben, dauerte ein Suchlauf zum Teil mehrere Minuten. Das Testsystem hatte einen mit 2,5 GHz getakteten 64-Bit-Prozessor mit 16 GB Arbeitsspeicher und 1 GB Level-1-Cache.

Zunächst betrachten wir die verwendete Prozessorzeit, wir verwenden hier den Begriff „Laufzeit“ synonym. Die Auswertung dieser Daten führen wir getrennt nach Art des Verfahrens (dynamisch, statisch on-line oder statisch off-line) durch. Als nächstes vergleichen wir dynamische und statische Verfahren und dort die on-line- bzw. off-line-Varianten. Abschließend analysieren wir die Anzahl der von den jeweiligen Algorithmen besuchten Knoten und Kanten.

Bei den Tests haben wir für jede Verfahrensklasse auch Daten für unsere Vergleichs-Implementation des Algorithmus von Dijkstra ohne Berücksichtigung von Abbiegeverboten erfasst. In den Grafiken ist der jeweilige Algorithmus mit „E“ bezeichnet.

Obwohl uns bei den statischen Verfahren auch eine bidirektionale Variante zur Verfügung gestanden hätte, haben wir die einfache unidirektionale Suche verwendet, um eine bessere Vergleichsmöglichkeit zwischen statischen und dynamischen Verfahren zu erhalten.

6.2.1 Dynamische Verfahren

Die verwendete Prozessorzeit der dynamischen Verfahren in Abhängigkeit von der Anzahl Knoten im Graphen wird in Abbildung 9 dargestellt. Dabei ist die Mehrfach-Knotenaufnahme mit „MeKnA“ und die Knoten- und Kantenaufnahme mit „KnKaA“ bezeichnet. Obwohl aus dieser Grafik für alle drei Algorithmen kein nennenswerter superlinearer Anstieg hergeleitet werden kann, zeigt das in Abbildung 10 dargestellte Verhältnis zwischen Laufzeit und Knotenanzahl die Superlinearität der Laufzeit deutlich auf.

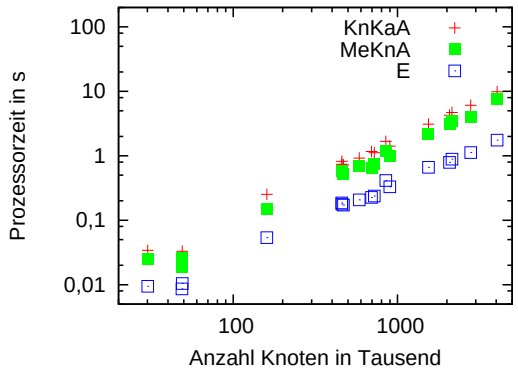


Abbildung 9: Verwendete Prozessorzeit bei den dynamischen Verfahren

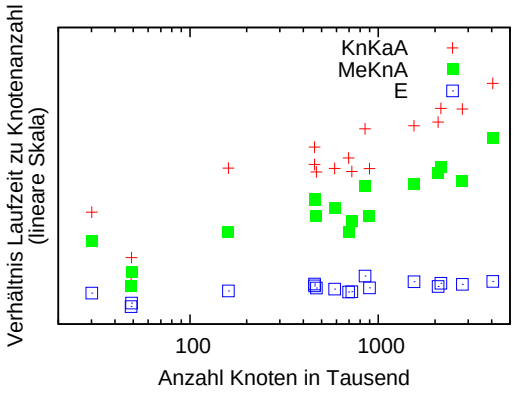


Abbildung 10: Verhältnis zwischen Laufzeit und Knotenanzahl bei den dynamischen Verfahren

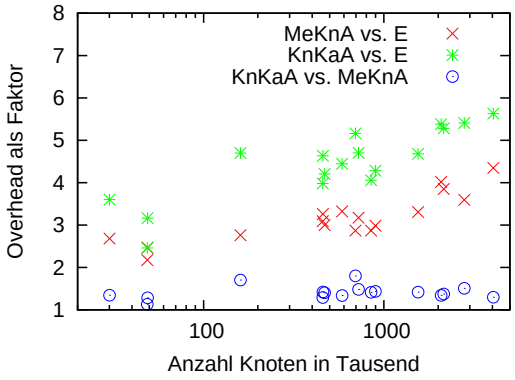


Abbildung 11: Vergleich der verwendeten Prozessorzeit bei den dynamischen Verfahren

In Abbildung 11 vergleichen wir die drei Verfahren paarweise. Die Implementation des Algorithmus von DIJKSTRA ohne Abbiegeverbote, die aus der LEDA-Bibliothek entnommen wurde, arbeitet wesentlich schneller als unsere Algorithmen. Das zeigt, dass in unserer Implementation noch Optimierungs-Potential steckt. Interessanterweise benötigt die Knoten- und Kantenaufnahme etwa anderthalbmal soviel Zeit wie die Mehrfach-Knotenaufnahme. Wir führen das zum einen auf eine nicht optimale Implementation der Domäne M der Prioritätswarteschlange zurück: Bei unserer Implementation passen die Elemente von M nicht in ein Maschinenwort. Zum anderen werden, wie wir später sehen werden, bei der Knoten- und Kantenaufnahme mehr Knoten und Kanten betrachtet als bei der Mehrfach-Knotenaufnahme.

6.2.2 Statische on-line-Verfahren

Abbildung 12 zeigt die Laufzeit der statischen on-line-Verfahren als Funktion der Knotenanzahl. Hier stellt „KaA“ die Kantenaufnahme und „KnS“ das Knotensplitting dar. Wie erwartet benötigt die Kantenaufnahme wesentlich mehr Zeit als die anderen Verfahren, der Overhead kann in Abbildung 13 abgelesen werden und liegt für den Vergleich zum Knotensplitting etwa zwischen 2,5 und 3. Dieser Wert könnte mit dem Verhältnis zwischen Kanten- und Knotenanzahl zusammenhängen, das haben wir an dieser Stelle allerdings nicht genauer untersucht.

Andererseits liegt der Overhead vom Knotensplitting im Vergleich zum einfachen Adapter bei vergleichsweise niedrigen 1,2. Das bestätigt unsere bei der Analyse des Knotensplittings in Unterabschnitt 4.5.6 aufgestellte Vermutung, dass die Laufzeit des Knotensplittings nur einen

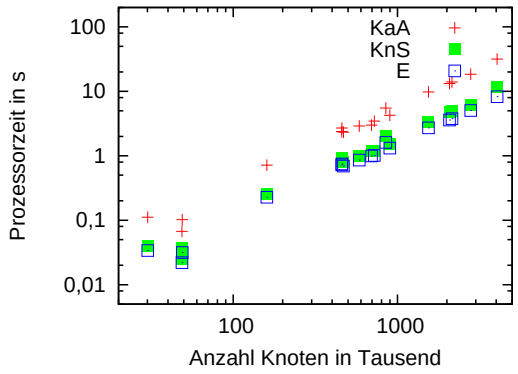


Abbildung 12: Verwendete Prozessorzeit bei den statischen on-line-Verfahren

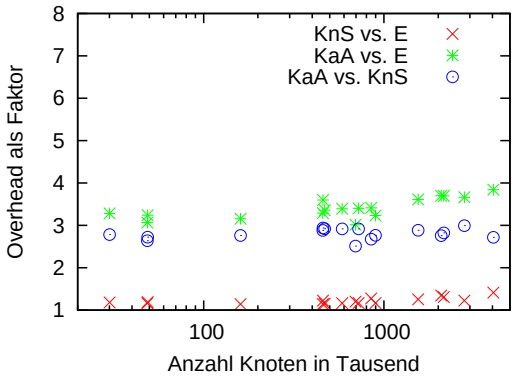


Abbildung 13: Vergleich der verwendeten Prozessorzeit bei den statischen on-line-Verfahren

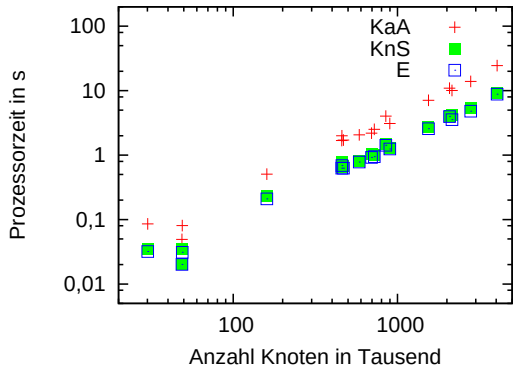


Abbildung 14: Verwendete Prozessorzeit bei den statischen off-line-Verfahren

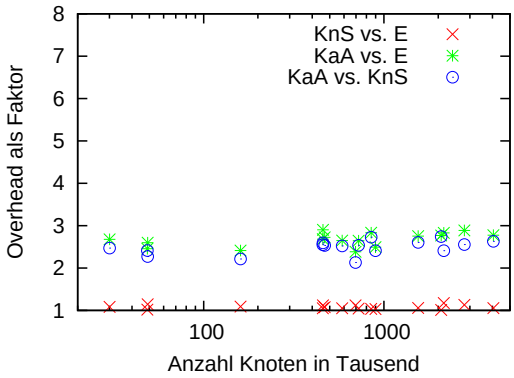


Abbildung 15: Vergleich der verwendeten Prozessorzeit bei den statischen off-line-Verfahren

Bruchteil höher ist als die des einfachen Algorithmus ohne Abbiegeverbote. Wir haben hier sehr faire Vergleichsbedingungen, da sowohl der einfache als auch der Knotensplitting-Adapter auf exakt dieselbe DIJKSTRA-Implementation aufsetzen.

6.2.3 Statische off-line-Verfahren

Die Abbildungen 14 und 15 zeigen dasselbe für den off-line-Fall. Hier ist bemerkenswert, dass der Overhead der Kantenaufnahme auf einen Wert um 2,5 gesunken ist. Auch der Overhead des Knotensplittings im Vergleich zum einfachen Adapter weist einen geringeren Wert auf.

6.2.4 Vergleich zwischen on-line- und off-line-Verfahren

In Abbildung 16 vergleichen wir on-line- mit off-line-Verfahren. Für den simplen Adapter ist der Overhead gering – die on-line-Variante tut im Wesentlichen genau dasselbe wie die off-line-Variante. (Für zwei Testfälle war die on-line-Variante sogar schneller als die off-line-Variante. Wir können nur vermuten, dass das mit der Größe der Graphen und damit verbundenen Cache-Effekten zusammenhängen kann.) Für das Knotensplitting arbeitet das on-line-Verfahren bei unserer Implementation etwa um Faktor 1,2 langsamer als das off-line-Verfahren. Eine extensive Optimierung des on-line-Knotensplittings könnte den Overhead weiter reduzieren. Ähnlich sieht es bei der Kantenaufnahme aus.

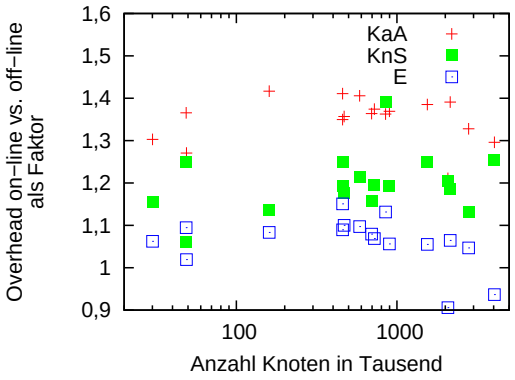


Abbildung 16: Vergleich zwischen on-line- und off-line-Verfahren

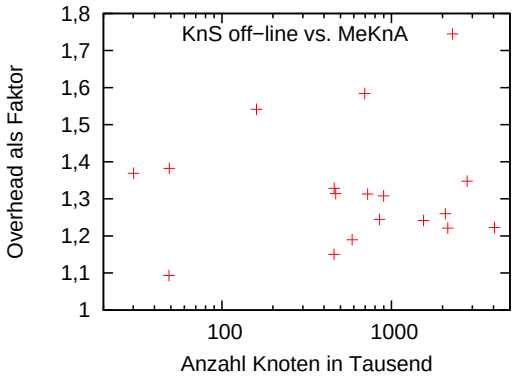


Abbildung 17: Vergleich eines statischen mit einem dynamischen Verfahren

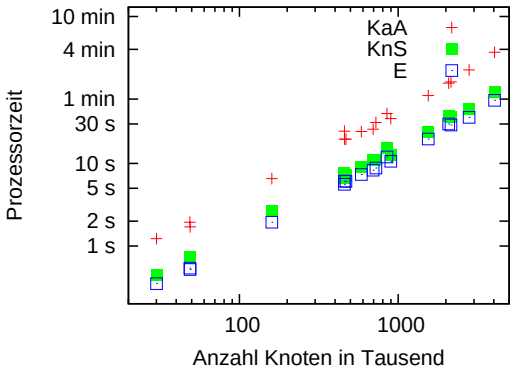


Abbildung 18: Verwendete Prozessorzeit bei der Konstruktion des Arbeitsgraphen bei den off-line-Verfahren

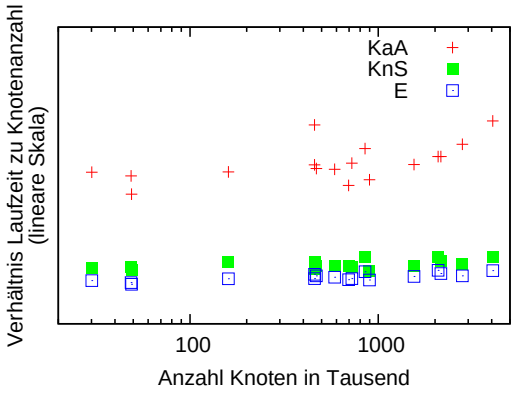


Abbildung 19: Verhältnis zwischen Laufzeit und Knotenanzahl bei der Konstruktion des Arbeitsgraphen bei den off-line-Verfahren

6.2.5 Vergleich zwischen dynamischen und statischen Verfahren

Als nächstes vergleichen wir in Abbildung 17 unser bestes statisches mit unserem besten dynamischen Verfahren: Die off-line-Version des Knotensplitting gegen die Mehrfach-Knotenaufnahme. Zu erwarten wäre, dass das statische off-line-Verfahren schneller als das dynamische Verfahren arbeitet: Das statische Verfahren macht nur eine Suche auf einem bereits existierenden Graph, während beim dynamischen Verfahren die Abbiegeverbote betrachtet werden müssen und die Suche insgesamt komplizierter ist. Erstaunlicherweise ist aber das statische Verfahren ca. anderthalbmal so langsam wie das dynamische.

Wir führen das darauf zurück, dass beim statischen Verfahren größerer Wert auf Allgemeinheit gelegt wurde. So ist die Suche bei allen statischen off-line-Verfahren nur ein Spezialfall eines on-line-Adapters. Wäre speziell für die off-line-Verfahren eine eigene Kürzeste-Wege-Such-Routine implementiert worden, würde der Vergleich vermutlich anders aussehen. Hinzu kommt, dass das dynamische Verfahren unter Verwendung eines Profilers nachoptimiert wurde. Außerdem werden auch beim Knotensplitting etwas mehr Knoten und Kanten betrachtet als bei der Mehrfach-Knotenaufnahme, wie wir später sehen werden.

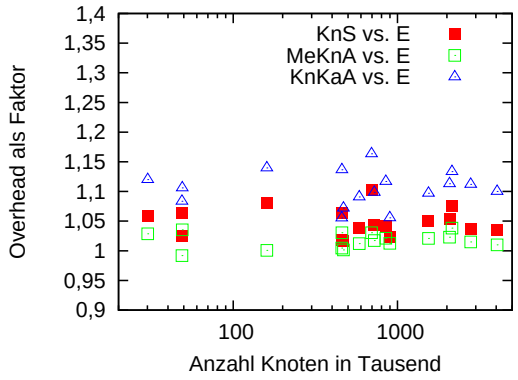


Abbildung 20: Vergleich der Anzahl betrachteter Knoten

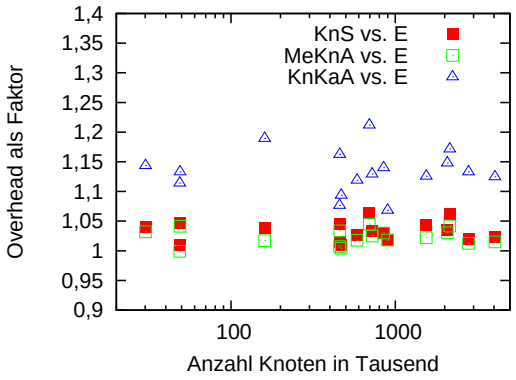


Abbildung 21: Vergleich der Anzahl betrachteter Kanten

6.2.6 Dauer der Vorverarbeitung bei statischen off-line-Verfahren

Wir schließen die Laufzeittests mit einer Analyse der Dauer der Vorverarbeitung bei den statischen off-line-Verfahren ab. Die Vorverarbeitung besteht bei diesen Verfahren aus dem Aufbau des Arbeitsgraphen per Tiefensuche. Das Diagramm in Abbildung 18 zeigt, dass die Dauer der Vorverarbeitung bei der Kantenaufnahme erwartungsgemäß eine Größenordnung größer als die des Knotensplittings ist. Der Aufbau des Arbeitsgraphen dauert allerdings auch beim Knotensplitting nur unwesentlich länger als die 1:1-Kopie per Tiefensuche beim Identitäts-Adapter: Selbst für den größten betrachteten Graphen wird nur knapp über einer Minute benötigt, trotz der nicht optimalen Implementation. Das dürfte ein für die Praxis akzeptabler Wert sein.

In Abbildung 19 wurde wieder die Laufzeit zur Knotenanzahl ins Verhältnis gesetzt. Als Ergebnis können wir festhalten, dass der Aufbau des Arbeitsgraphen linear in der Größe des Ausgangsgraphen ist.

6.2.7 Anzahl betrachteter Knoten und Kanten

Zusätzlich zu den reinen Laufzeittests hat unsere Anwendung noch die Anzahl betrachteter Knoten und Kanten erfasst. In den Abbildungen 20 und 21 ist der Overhead im Vergleich zum Algorithmus von Dijkstra dargestellt. Die Mehrfach-Knotenaufnahme betrachtet insgesamt weniger Knoten und Kanten als das Knotensplitting, was wir auf die exaktere Update-Prozedur der Prioritätswarteschlange zurückführen. Die Knoten- und Kantenaufnahme schneidet hingegen unerwartet schlecht ab – das könnte daran liegen, dass unsere Implementation leicht vom hergeleiteten Algorithmus abweicht. Wir untersuchen hier das Verfahren Kantenaufnahme nicht genauer, da hier der Overhead weit über dem der anderen drei Verfahren liegt.

7 Ergebnisse und Ausblick

Wir haben in dieser Arbeit mehrere Herangehensweisen an das Problem der Kürzesten-Wege-Suche mit Berücksichtigung von Abbiegeverboten untersucht und verglichen. Trotz des etwas schlechteren Abschneidens bei den Laufzeittests favorisieren wir die statischen Verfahren, da sie besser in bestehende Umgebungen und mit Beschleunigungstechniken integriert werden können. Selbst wenn im statischen Fall aus Effizienzgründen ein off-line-Verfahren verwendet wird, kann für die von uns vorgestellten Verfahren die on-line-Variante verwendet werden, um einen off-line-Adapter zu erzeugen. Dadurch ist später gegebenenfalls ein Umstieg auf die on-line-Variante möglich, beispielsweise falls der Graph zu häufig geändert wird. Für Anwendungen, die bisher nicht oder nur eingeschränkt mit Abbiegeverboten umgehen, können wir als Sofortlösung das Knotensplitting empfehlen, das

mit den hier vorgestellten Methoden gut in jede Art von Kürzeste-Wege-Anwendung eingepasst werden kann.

Zu untersuchen wäre, welchen Einfluss die Berücksichtigung von Abbiegeverboten auf die Effizienz der Beschleunigungstechniken hat. Ein bei einem statischen Verfahren entstehender Arbeitsgraph kann durchaus ganz andere Eigenschaften als der zugrunde liegende Ausgangsgraph haben, so dass die Wirksamkeit der Beschleunigungstechniken nicht mehr unbedingt gewährleistet ist.

Ein anderer Aspekt, den wir hier nicht betrachtet haben, ist die Auswirkung von Änderungen am Ausgangsgraphen oder an den Abbiegeverboten auf den Arbeitsgraphen bei einem statischen Verfahren. Eine kleine Änderung an den Abbiegeverboten löst möglicherweise weit reichende Änderungen im Arbeitsgraphen aus. Wenn ein Algorithmus über Änderungen im Graphen informiert werden möchte, beispielsweise um seine vorberechnete Information zu aktualisieren, muss ein statischer on-line-Adapter Änderungen am Ausgangsgraphen oder an den Abbiegeverboten korrekt in Änderungen auf dem Arbeitsgraphen transformieren.

Schlussendlich müssen wir die Frage der konkreten Realisierbarkeit effizienter on-line-Transformationen für allgemeine Wegeverbote offen lassen.

A Pseudocodes der beschriebenen Algorithmen

Eingabe: Gewichteter gerichteter Graph mit nichtnegativer Kostenfunktion $G(V, E, c)$
Eingabe: Startknoten $v_s \in V$
Eingabe: Zielknoten $v_t \in V$

Ausgabe: Kürzester Pfad P von v_s zu v_t

Daten: Prioritätswarteschlange $Q \subset \mathbb{R}_+ \times V$
Daten: Vorgängerkantenvektor $P_s : V \to E$
Daten: Kostenvektor $C_s : V \to \mathbb{R}_+$

```
1  Beginn
2      für alle v ∈ V tue
3          C_s[v] ← ∞; P_s[v] ← nil;
4      Ende
5      Q ← {(0, v_s)}; C_s[v_s] ← 0;
10     wiederhole
11         (c_min, v_min) ← extract_min(Q);
20         wenn v_min = v_t dann
               // Ziel erreicht
21             v ← v_t;
22             solange v ≠ v_s tue
23                 e ← P_s[v];
24                 v ← source(e);
25                 P ← e + P;
26             Ende
27             zurück P
28         Ende
30         für alle e_neu ∈ out(v_min) tue
31             v_neu ← target(e_neu);
32             c_neu ← c_min + c(e_neu);
40             wenn c_neu < C_s[v_neu] dann
41                 wenn C_s[v_neu] < ∞ dann  decrease_key(Q, (c_neu, v_neu));
42                 sonst  insert(Q, (c_neu, v_neu));
43                 C_s[v_neu] ← c_neu; P_s[v_neu] ← e_neu;
44             Ende
50         Ende
51     bis (|Q| = 0);
52     Fehler: Es gibt keinen Pfad von v_s nach v_t;
53 Ende
```

Abbildung 22: Der Algorithmus von DIJKSTRA

Eingabe: Gewichteter gerichteter Graph mit nichtnegativer Kostenfunktion $G(V, E, c)$
Eingabe: Startknotenmenge $V_s \subset V$
Eingabe: Zielknotenmenge $V_t \subset V$

Ausgabe: Kürzester Pfad P von einem Knoten aus V_s zu einem Knoten aus V_t

Daten: Prioritätswarteschlange $Q \subset \mathbb{R}_+ \times V$
Daten: Vorgängerkantenvektor $P_s : V \to E$
Daten: Kostenvektor $C_s : V \to \mathbb{R}_+$

```
1  Beginn
2      für alle v ∈ V tue
3          C_s[v] ← ∞; P_s[v] ← nil;
4      Ende
5      für alle v ∈ V_s tue  Q ← {(0, v)}; C_s[v] ← 0;
10     wiederhole
11         (c_min, v_min) ← extract_min(Q);
20         wenn v ∈ V_t dann
               // Ziel erreicht
21             v ← v_t;
22             solange v ≠ v_s tue
23                 e ← P_s[v];
24                 v ← source(e);
25                 P ← e + P;
26             Ende
27             zurück P
28         Ende
30         für alle e_neu ∈ out(v_min) tue
31             v_neu ← target(e_neu);
32             c_neu ← c_min + c(e_neu);
40             wenn c_neu < C_s[v_neu] dann
41                 wenn C_s[v_neu] < ∞ dann  decrease_key(Q, (c_neu, v_neu));
42                 sonst  insert(Q, (c_neu, v_neu));
43                 C_s[v_neu] ← c_neu; P_s[v_neu] ← e_neu;
44             Ende
50         Ende
51     bis (|Q| = 0);
52     Fehler: Es gibt keinen Pfad von einem Knoten aus V_s zu einem Knoten aus V_t;
53 Ende
```

Abbildung 23: Der Algorithmus von DIJKSTRA zur Lösung des MSTSP

Eingabe: Gewichteter gerichteter Graph mit nichtnegativer Kostenfunktion $G(V, E, c)$
Eingabe: Abbiegeverbote $T \subset E \times E$
Eingabe: Startknoten $v_s \in V$
Eingabe: Zielknoten $v_t \in V$

Ausgabe: Kürzester Pfad P von v_s zu v_t unter Berücksichtigung der Abbiegeverbote

Daten: Prioritätswarteschlange $Q \subset \mathbb{R}_+ \times (V \times 2^E \times E)$
Daten: Vorgängerkantenvektor $P_s : E \to E$
Daten: ~~Kostenvektor $C_s : V \to \mathbb{R}_+$~~

1 **Beginn**
2 **für alle** $e \in E$ **tue**
3 ~~$C_s[v] \leftarrow \infty$;~~ $P_s[e] \leftarrow nil$;
4 **Ende**
5 $Q \leftarrow \{(0, (v_s, out(v_s), nil))\}$;
10 **wiederhole**
11 $(c_{min}, (v_{min}, A_{min}, e_{min})) \leftarrow extract\_min(Q)$;
20 **wenn** $v_{min} = v_t$ **dann**
`// Ziel erreicht`
21 **solange** $e_{min} \neq nil$ **tue**
22 $P \leftarrow e_{min} + P$;
23 $e_{min} \leftarrow P_s[e_{min}]$;
24 **Ende**
25 **zurück** P
26 **Ende**
27 $E_{besucht} \leftarrow E_{besucht} \cup A_{min}$;
30 **für alle** $e_{neu} \in A_{min}$ **tue**
31 $v_{neu} \leftarrow target(e_{neu})$;
32 $c_{neu} \leftarrow c_{min} + c(e_{neu})$;
33 $Q_{v_{neu}} \leftarrow \sigma_{v=v_{neu}}(Q)$;
34 $Q^-_{v_{neu}} \leftarrow \sigma_{c \leq c_{neu}}(Q_{v_{neu}})$;
35 $Q^+_{v_{neu}} \leftarrow \sigma_{c > c_{neu}}(Q_{v_{neu}})$;
36 $A_{neu} \leftarrow out(v_{neu}) \setminus [(forbidden(e_{neu})) \cup E_{besucht} \cup (\bigcup \pi_3(Q^-_{v_{neu}}))]$;
40 **wenn** $A_{neu} \neq \emptyset \vee v = v_t$ **dann**
41 $Q^{++}_{v_{neu}} \leftarrow \{(c, (v, A \setminus A_{neu}, e)) : (c, (v, A, e)) \in Q^+_{v_{neu}}\}$;
42 $Q \leftarrow \sigma_{A \neq \emptyset \vee v = v_t}(Q \cup \{(c_{neu}, (v_{neu}, A_{neu}, e_{neu}))\} \setminus Q^+_{v_{neu}} \cup Q^{++}_{v_{neu}})$;
43 ~~$C_s[v_{neu}] \leftarrow c_{neu}$;~~ $P_s[e_{neu}] \leftarrow e_{min}$;
44 **Ende**
50 **Ende**
51 **bis** $(|Q| = 0)$;
52 Fehler: Es gibt keinen Pfad ohne Abbiegeverbote von v_s nach v_t;
53 **Ende**

Abbildung 24: Der Algorithmus von DIJKSTRA mit mehrfacher Knotenaufnahme

Eingabe: Gewichteter gerichteter Graph mit nichtnegativer Kostenfunktion $G(V, E, c)$
Eingabe: Abbiegeverbote $T \subset E \times E$
Eingabe: Startknoten $v_s \in V$
Eingabe: Zielknoten $v_t \in V$

Ausgabe: Kürzester Pfad P von v_s zu v_t unter Berücksichtigung der Abbiegeverbote

Daten: Domäne $M := V \uplus E$
Daten: Prioritätswarteschlange $Q \subset \mathbb{R}_+ \times M$
Daten: Vorgänger-Abbildung $P_s : M \to M$
Daten: Kostenvektor $C_s : M \to \mathbb{R}_+$

```
1  Beginn
2      für alle m ∈ M tue
3          C_s[m] ← ∞; P_s[m] ← (nil, nil);
4      Ende
5      Q ← {(0, v_s)}; C_s[v_s] ← 0;
10     wiederhole
11         (c_min, m_min) ← extract_min(Q);
12         v_min ← k(m_min);
20         wenn v_min = v_t dann
               // Ziel erreicht
21             m ← m_min;
22             solange m ≠ v_s tue
23                 P ← (k(P_s[m]), k(m)) +P;
24                 m ← P_s[m];
25             Ende
26             zurück P
27         Ende
28         wenn m_min ∈ V dann  F ← ∅;
29         sonst  F ← forbidden(m_min);
30         für alle e_neu ∈ out(v_min) \ F tue
31             ~~v_neu ← target(e_neu)~~;
32             c_neu ← c_min + c(e_neu);
33             m_neu = h(e_neu);
40             wenn c_neu < C_s[m_neu] dann
41                 wenn C_s[m_neu] < ∞ dann  decrease_key(Q, (c_neu, m_neu));
42                 sonst  insert(Q, (c_neu, m_neu));
43                 C_s[m_neu] ← c_neu; P_s[m_neu] ← m_min;
44             Ende
50         Ende
51     bis (|Q| = 0);
52     Fehler: Es gibt keinen Pfad ohne Abbiegeverbote von v_s nach v_t;
53 Ende
```

Abbildung 25: Der Algorithmus von DIJKSTRA mit Knoten- und Kantenaufnahme

B Ablaufbeispiel für die Mehrfach-Knotenaufnahme

Wir möchten die Arbeitsweise des Algorithmus, insbesondere der Aktualisierung der Prioritätswarteschlange, an einem Beispiel vorstellen.

Gegeben sei der Graph $G(V, E, c)$ mit $V = \{v_1, v_2, \ldots, v_6\}$ und folgender Kantenmenge E und Kostenfunktion c:

e	$source(e)$	$target(e)$	$c(e)$
e_1	v_1	v_3	2
e_2	v_1	v_3	3
e_3	v_1	v_2	1
e_4	v_2	v_3	2
e_5	v_3	v_4	2
e_6	v_3	v_5	2
e_7	v_3	v_6	2

Ferner seien die Abbiegeverbote $T := \{(e_1, e_5), (e_2, e_6), (e_4, e_7)\}$ definiert. Abbildung 26 zeigt den Graphen.

Das Beispiel spiegelt nicht gerade reale Gegebenheiten von Straßengraphen wider. Insbesondere gilt für die Knoten v_4, v_5 und v_6 unsere Forderung nach $outdeg(v) > 0$ nicht. Die Arbeitsweise des Algorithmus wird aber gut illustriert.

Gesucht sei der kürzeste Weg zwischen v_1 und v_4. Offenbar muss dieser über Kante e_5 und Knoten v_3 führen. Der günstigste Weg, v_3 zu erreichen, ist über die Kante e_1. Von Kante e_1 geht aber ein Abbiegeverbt nach e_5 aus, so dass nur Wege über e_2 oder e_4 in Frage kommen. Entsprechend sind $[e_2, e_5]$ oder $[e_3, e_4, e_5]$ kürzeste Wege unter Berücksichtigung der Abbiegeverbote. Die minimalen Wegkosten betragen 5.

Wir geben jeweils an, welche Zeile im Pseudocode abgearbeitet wird und welchen Wert die gelesenen und geschriebenen Werte haben. Die Spalte „#“ gibt eine fortlaufende Nummerierung der Aktionen an. Die rechte Spalte enthält Zeiger auf den Ursprung der für die aktuelle Zeile benötigten Daten. Der Zeiger ist als gestrichelte Linie ausgeführt, wenn das Ursprungsdatum außerhalb der inneren Schleife berechnet wird und somit innerhalb der inneren Schleife eine Konstante, also „weniger aktuell“, ist. Aktionen, bei denen nichts Interessantes passiert, wurden teilweise ausgelassen.

Tabelle 1 zeigt den ersten Durchlauf der Hauptschleife. Nach dem ersten Durchlauf enthält die Prioritätswarteschlange zwei Einträge für den Knoten v_3, da die Kante e_5 aufgrund eines Abbiegeverbotes nur über den Umweg e_2 erreicht werden kann. Dabei wurde in Aktion 28 die Ausgangskantenmenge A_{neu} für die an zweiter Stelle betrachtete Kante e_2 durch den bereits in Q befindlichen Eintrag beschnitten. Wenn die Kante e_2 vor Kante e_1 betrachtet worden wäre, würde das Einfügen des Eintrags zu e_1 den bereits in Q befindlichen Eintrag zu e_2 modifizieren, so dass am Ende der Zustand der Prioritätswarteschlange derselbe ist wie bei der von uns präsentierten Reihenfolge. Die Arbeitsweise des Algorithmus ist an dieser Stelle unabhängig von der Betrachtungs-Reihenfolge der Ausgangskanten eines Knotens.

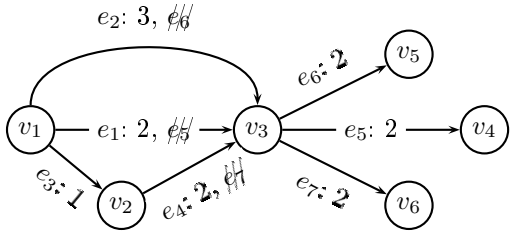


Abbildung 26: Beispiel-Graph zur Demonstration der Mehrfach-Knotenaufnahme

Der zweite Durchlauf der Hauptschleife wird in Tabelle 2 dargestellt. Hier ist A_{neu} am Ende leer, da der neu betrachtete Weg über v_2 für keine Zielkante geringere Kosten mit sich bringt. Entsprechend wird die Prioritätswarteschlange gar nicht aktualisiert, da v_3 nicht das Ziel ist.

Beim dritten Durchlauf (Tabelle 3) werden die Knoten v_5 und v_6 erreicht. Da sie keine Ausgangskanten haben, ist A_{neu} jeweils leer. Da beide Knoten nicht das Ziel darstellen, wird die Prioritätswarteschlange nicht aktualisiert.

Der vierte Durchlauf, dargestellt in Tabelle 4, erreicht endlich den Zielknoten v_4. Obwohl er keine Ausgangskanten hat, wird er trotzdem in Q eingefügt. Hier wird auch zum ersten Mal P_s sinnvoll aktualisiert (Aktion 105).

Die Ermittlung des kürzesten Pfads erfolgt im fünften Durchlauf. Der Algorithmus findet hier entsprechend der Vorgängerliste P_s den Pfad $[e_2, e_5]$, der bei letzten Aktion 118 zurückgegeben wird. Der andere kürzeste Pfad wird unter keinen Umständen gefunden, da Kante e_2 immer vor Kante e_4 betrachtet wird.

#	Zeile	Variable	Wert	
1	4	P_s	$\{e_1, e_2, e_3, e_4, e_5, e_6, e_7 \to nil\}$	
2	5	Q	$\{(0, (v_1, \{e_1, e_2, e_3\}, nil))\}$	
3	11	c_{min}	0	
4	11	v_{min}	v_1	
5	11	A_{min}	$\{e_1, e_2, e_3\}$	
6	11	e_{min}	nil	
7	11	Q	$\emptyset$	
8	27	$E_{besucht}$	$\{e_1, e_2, e_3\}$	
9	30	e_{neu}	e_1	
10	31	v_{neu}	v_3	
11	32	c_{neu}	2	
12	33	$Q_{v_{neu}}$	$\emptyset$	
13	34	$Q^-_{v_{neu}}$	$\emptyset$	
14	35	$Q^+_{v_{neu}}$	$\emptyset$	
15	36	A_{neu}	$\{e_5, e_6, e_7\} \setminus [(\{e_5\} \cup \{e_1, e_2, e_3\} \cup \emptyset)]$	
16	36	A_{neu}	$\{e_6, e_7\}$	
17	41	$Q^{++}_{v_{neu}}$	$\emptyset$	
18	42	Q	$\sigma_{A \neq \emptyset \vee v = v_t}(Q \cup \{(2, (v_3, \{e_6, e_7\}, e_1))\} \setminus \emptyset \cup \emptyset)$	
19	42	Q	$\sigma(\{(2, (v_3, \{e_6, e_7\}, e_1))\})$	
20	42	Q	$\{(2, (v_3, \{e_6, e_7\}, e_1))\}$	
21	43	P_s	$\{e_1, e_2, e_3, e_4, e_5, e_6, e_7 \to nil\}$	
22	30	e_{neu}	e_2	
23	31	v_{neu}	v_3	
24	32	c_{neu}	3	
25	33	$Q_{v_{neu}}$	$\{(2, (v_3, \{e_6, e_7\}, e_1))\}$	
26	34	$Q^-_{v_{neu}}$	$\{(2, (v_3, \{e_6, e_7\}, e_1))\}$	
27	35	$Q^+_{v_{neu}}$	$\emptyset$	
28	36	A_{neu}	$\{e_5, e_6, e_7\} \setminus [(\{e_6\} \cup \{e_1, e_2, e_3\} \cup \{e_6, e_7\})]$	
29	36	A_{neu}	$\{e_5\}$	
30	41	$Q^{++}_{v_{neu}}$	$\emptyset$	
31	42	Q	$\sigma_{A \neq \emptyset \vee v = v_t}(Q \cup \{(3, (v_3, \{e_5\}, e_2))\} \setminus \emptyset \cup \emptyset)$	
32	42	Q	$\sigma(\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\})$	
33	42	Q	$\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
34	43	P_s	$\{e_1, e_2, e_3, e_4, e_5, e_6, e_7 \to nil\}$	
35	30	e_{neu}	e_3	
36	31	v_{neu}	v_2	
37	32	c_{neu}	1	
38	33	$Q_{v_{neu}}$	$\emptyset$	
39	34	$Q^-_{v_{neu}}$	$\emptyset$	
40	35	$Q^+_{v_{neu}}$	$\emptyset$	
41	36	A_{neu}	$\{e_4\} \setminus [(\emptyset \cup \{e_1, e_2, e_3\} \cup \emptyset)]$	
42	36	A_{neu}	$\{e_4\}$	
43	41	$Q^{++}_{v_{neu}}$	$\emptyset$	
44	42	Q	$\sigma_{A \neq \emptyset \vee v = v_t}(Q \cup \{(1, (v_2, \{e_4\}, e_3))\} \setminus \emptyset \cup \emptyset)$	
45	42	Q	$\sigma(\{(1, (v_2, \{e_4\}, e_3)), (2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\})$	
46	42	Q	$\{(1, (v_2, \{e_4\}, e_3)), (2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
47	43	P_s	$\{e_1, e_2, e_3, e_4, e_5, e_6, e_7 \to nil\}$	

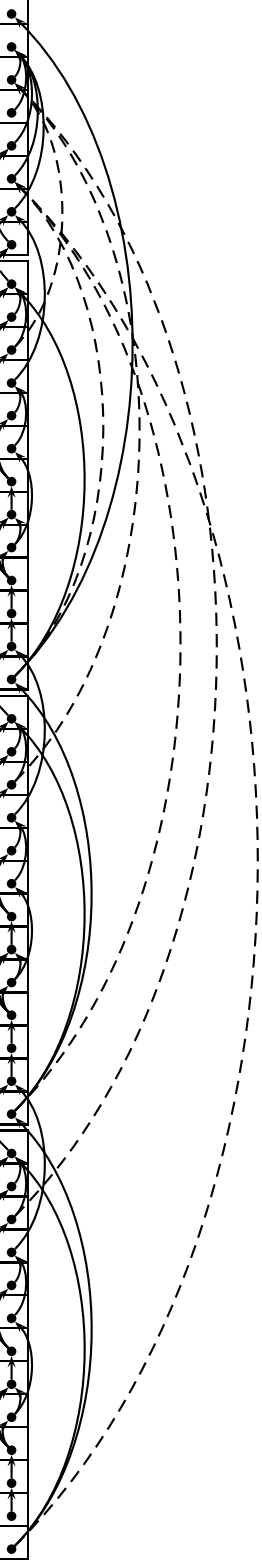


Tabelle 1: Erster Durchlauf der Hauptschleife bei der Mehrfach-Knotenaufnahme

#	Zeile	Variable	Wert	
49	[27]	$E_{besucht}$	$\{e_1, e_2, e_3\}$	
50	[42]	Q	$\{(1, (v_2, \{e_4\}, e_3)), (2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
51	11	c_{min}	1	
52	11	v_{min}	v_2	
53	11	A_{min}	$\{e_4\}$	
54	11	e_{min}	e_3	
55	11	Q	$\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
56	27	$E_{besucht}$	$\{e_1, e_2, e_3, e_4\}$	
57	30	e_{neu}	e_4	
58	31	v_{neu}	v_3	
59	32	c_{neu}	3	
60	33	$Q_{v_{neu}}$	$\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
61	34	$Q^-_{v_{neu}}$	$\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
62	35	$Q^+_{v_{neu}}$	$\emptyset$	
63	36	A_{neu}	$\{e_5, e_6, e_7\} \setminus [(\{e_7\} \cup \{e_1, e_2, e_3, e_4\} \cup \{e_5, e_6, e_7\})]$	
64	36	A_{neu}	$\emptyset$	

Tabelle 2: Zweiter Durchlauf der Hauptschleife bei der Mehrfach-Knotenaufnahme

#	Zeile	Variable	Wert	
66	[11]	Q	$\{(2, (v_3, \{e_6, e_7\}, e_1)), (3, (v_3, \{e_5\}, e_2))\}$	
67	[27]	$E_{besucht}$	$\{e_1, e_2, e_3, e_4\}$	
68	11	c_{min}	2	
69	11	v_{min}	v_3	
70	11	A_{min}	$\{e_6, e_7\}$	
71	11	e_{min}	e_1	
72	11	Q	$\{(3, (v_3, \{e_5\}, e_2))\}$	
73	27	$E_{besucht}$	$\{e_1, e_2, e_3, e_4, e_6, e_7\}$	
74	30	e_{neu}	e_6	
75	31	v_{neu}	v_5	
76	32	c_{neu}	4	
77	33	$Q_{v_{neu}}$	$\emptyset$	
78	34	$Q^-_{v_{neu}}$	$\emptyset$	
79	35	$Q^+_{v_{neu}}$	$\emptyset$	
80	36	A_{neu}	$\emptyset \setminus \ldots$	
81	36	A_{neu}	$\emptyset$	
82	30	e_{neu}	e_7	
83	31	v_{neu}	v_6	
			$\vdots$	
85	36	A_{neu}	$\emptyset$	

Tabelle 3: Dritter Durchlauf der Hauptschleife bei der Mehrfach-Knotenaufnahme

#	Zeile	Variable	Wert	
87	[11]	Q	$\{(3,(v_3,\{e_5\},e_2))\}$	
88	[27]	$E_{besucht}$	$\{e_1,e_2,e_3,e_4,e_6,e_7\}$	
89	[43]	P_s	$\{e_1,e_2,e_3,e_4,e_5,e_6,e_7 \rightarrow nil\}$	
90	11	c_{min}	3	
91	11	v_{min}	v_3	
92	11	A_{min}	$\{e_5\}$	
93	11	e_{min}	e_2	
94	11	Q	$\emptyset$	
95	27	$E_{besucht}$	E	
96	30	e_{neu}	e_5	
97	31	v_{neu}	v_4	
98	32	c_{neu}	5	
			$\vdots$	
100	36	A_{neu}	$\emptyset$	
101	41	$Q^{++}_{v_{neu}}$	$\emptyset$	
102	42	Q	$\sigma_{A\neq\emptyset\vee v=v_t}\,(Q \cup \{(5,(v_4,\emptyset,e_5))\} \setminus \emptyset \cup \emptyset)$	
103	42	Q	$\sigma\,(\{(5,(v_4,\emptyset,e_5))\})$	
104	42	Q	$\{(5,(v_4,\emptyset,e_5))\}$	
105	43	P_s	$\{e_5 \rightarrow e_2; e_1,e_2,e_3,e_4,e_6,e_7 \rightarrow nil\}$	

Tabelle 4: Vierter Durchlauf der Hauptschleife bei der Mehrfach-Knotenaufnahme

#	Zeile	Variable	Wert	
107	[42]	Q	$\{(5,(v_4,\emptyset,e_5))\}$	
108	[43]	P_s	$\{e_5 \rightarrow e_2; e_1,e_2,e_3,e_4,e_6,e_7 \rightarrow nil\}$	
109	11	c_{min}	5	
110	11	v_{min}	v_4	
111	11	A_{min}	$\emptyset$	
112	11	e_{min}	e_5	
113	11	Q	$\emptyset$	
114	22	P	$[e_5]$	
115	23	e_{min}	e_2	
116	22	P	$[e_2,e_5]$	
117	23	e_{min}	nil	
118	25	P	$[e_2,e_5]$	
Ende				

Tabelle 5: Fünfter Durchlauf der Hauptschleife bei der Mehrfach-Knotenaufnahme

C Grammatik des GPR-Formats in EBNF

Hier stellen wir kurz das Dateiformat vor, das wir für unsere Referenzimplementation verwendet haben. In Schreibmaschinenschrift gesetzte Symbole sind Nichtterminale (Kleinbuchstaben, z.B. `goal`) bzw. Terminale (Großbuchstaben, z.B. `STRING`) der Grammatik. In die Grammatik eingebettete Terminale sind in proportionaler Fettschrift gesetzt (z.B. **name:**). Wir verzichten bei den Terminalsymbolen auf eine Darstellung als reguläre Ausdrücke.

```
goal           ::=  name lines
name           ::=
name           ::=  name: STRING NEWLINE
lines          ::=
lines          ::=  lines line
line           ::=  line_decl NEWLINE
line_decl      ::=
line_decl      ::=  EDGE_ID length : NODE_ID -> NODE_ID restriction
length         ::=
length         ::=  = DOUBLE
restriction    ::=
restriction    ::=  # edge_id_list
edge_id_list   ::=  EDGE_ID edge_id_list2
edge_id_list2  ::=
edge_id_list2  ::=  , edge_id_list

STRING         ::=  [Durch Gänsefüßchen begrenzte Zeichenkette]
NEWLINE        ::=  [Zeilenumbruch]
EDGE_ID        ::=  [Ziffernfolge mit vorangestelltem e]
NODE_ID        ::=  [Ziffernfolge mit vorangestelltem n]
DOUBLE         ::=  [Fließkommazahl mit . als Dezimaltrennzeichen]
```

Leerzeichen zwischen Nichtterminalen sind erlaubt. Kommentare im C++-Stil (//) sind ebenfalls erlaubt: Der Text hinter den zwei Schrägstrichen bis zum Zeilenende wird ignoriert. Leerzeilen werden ebenfalls ignoriert.

Jedes Nichtterminal `line_decl` gibt eine Kante an, die dem Graphen hinzugefügt wird. Die zu erzeugenden Knoten werden nicht vorab aufgelistet, sie ergeben sich aus der Definition der Kanten. Das Nichtterminal `restriction` gibt eine optionale Liste der an, welche Kanten ausgehend von der aktuell in `line_decl` betrachteten Kante verboten sind.

Es folgt der Beispiel-Graph aus Anhang B im GPR-Format, wobei das v bei Knoten wegen der Voraussetzungen für das Format durch ein `n` ersetzt wurde.

```
name: "Beispiel-Graph aus Anhang B"

e1 = 2: n1 -> n3 # e5
e2 = 3: n1 -> n3 # e6 // Multikante
e3 = 1: n1 -> n2
e4 = 2: n2 -> n3 # e7
e5 = 2: n3 -> n4      // Der Graph ist nicht stark zusammenhängend
e6 = 2: n3 -> n5
e7 = 2: n3 -> n6
```

Literatur

[AHU83] Aho, Alfred V. ; Hopcroft, John E. ; Ullman, Jeffrey D.: *Data Structures and Algorithms.* Addison-Wesley, 1983

[CLRS01] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms.* Second edition. The MIT Press and McGraw-Hill, 2001

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

[Flo62] Floyd, Robert W.: Algorithm 97: Shortest path. In: *Commun. ACM* 5 (1962), Nr. 6, S. 345

[GKW05] Goldberg, Andrew V. ; Kaplan, Haim ; Werneck, Renato F.: Reach for A^*: Efficient Point-to-Point Shortest Path Algorithms / Microsoft Research. Redmond, Washington, 2005. – Technical Report MSR-TR-2005-132

[Gra] `http://graphml.graphdrawing.org/`

[Gut04] Gutman, Ronald J.: Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In: *ALENEX/ANALC*, 2004, S. 100–111

[Her92] Herold, Helmut: *lex und yacc.* Addison Wesley, 1992

[HNR68] Hart, Peter E. ; Nilsson, Nils J. ; Raphael, Bertram: A formal basis for the heuristic determination of minimum cost paths. In: *IEEE transactions on systems science and cybernetics*, 1968, S. 100–107

[Hol03] Holzer, Martin: Hierarchical speed-up techniques for shortest-path algorithms / Universität Konstanz, Fachbereich Informatik. 2003. – Forschungsbericht

[Joh77] Johnson, Donald B.: Efficient Algorithms for Shortest Paths in Sparse Networks. In: *J. ACM* 24 (1977), Nr. 1, S. 1–13. – ISSN 0004–5411

[LL95] Lang, Stefan M. ; Lockemann, Peter C.: *Datenbankeinsatz.* 1. Auflage. Springer, 1995

[MR95] Motwani, Rajeev ; Raghavan, Prabhakar: *Randomized Algorithms.* Cambridge University Press, 1995

[NM99] Näher, Stefan ; Mehlhorn, Kurt: *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999. – `http://www.algorithmic-solutions.com/`

[Poh69] Pohl, Ira: Bi-directional and heuristic search in path problems / Stanford Linear Accelerator Center. Stanford, California, 1969. – Technical report 104

[SB77] Schaechterle, K. ; Braun, J.: Vergleichende Untersuchungen vorhandener Verfahren für Verkehrsumlegungen unter Verwendung elektronischer Rechenanlagen. In: *Forschung Straßenbau und Straßenverkehrstechnik* Heft 222 (1977)

[Sch00] Schmid, Wolfgang: *Berechnung kürzester Wege in Straßennetzen mit Wegeverboten*, Universität Stuttgart, Diss., 2000

[Sch05a] Schultes, Dominik: *Fast and Exact Shortest Path Queries Using Highway Hierarchies*, Universität des Saarlandes, Master-Arbeit, 2005

[Sch05b] Schütz, Birk: *Partition-Based Speed-Up of* Dijkstra*'s Algorithm*, Universität Karlsruhe, Studienarbeit, 2005

[Sed88] SEDGEWICK, Robert: *Algorithms*. Second edition. Addison-Wesley, 1988

[SS05] SANDERS, Peter ; SCHULTES, Dominik: Highway Hierarchies Hasten Exact Shortest Path Queries. In: *13th European Symposium on Algorithms (ESA)*, 2005 (LNCS). – Noch nicht erschienen

[Wil04] WILLHALM, Thomas: *Von Java nach C++*. 2004. – S. 45–53. `http://www.inf.uni-konstanz.de/algo/lehre/skripte/Quellen/java2c++.pdf`

[WW03] WAGNER, Dorothea ; WILLHALM, Thomas: Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In: *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)* Bd. 2832, Springer-Verlag, 2003, S. 776–787

[WW05] WILLHALM, Thomas ; WAGNER, Dorothea: Shortest Path Speedup Techniques. In: *Algorithmic Methods for Railway Optimization*, 2005 (LNCS). – Noch nicht erschienen