Vereinheitlichte Darstellung von Techniken zur effizienten Kürzeste-Wege-Suche

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart
zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Stefan Lewandowski

aus Northeim

Hauptberichter:	Prof. Dr. Volker Claus
Mitberichter:	Prof. Dr. Ulrich Hertrampf
Tag der mündlichen Prüfung:	29. September 2004

Stuttgart, 16. Juni 2005

Danksagung

Diese Arbeit entstand im Rahmen meiner Tätigkeit am Lehrstuhl Formale Konzepte der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart. Ich danke meinem Kollegen Wolfgang Schmid für die Diskussionen zum Thema der Arbeit. Desweiteren möchte ich insbesondere meinem Bruder Achim sowie Cerstin Mahlow für das Korrekturlesen danken. Meiner Frau Miriam und meiner Mutter danke ich für den seelischen Beistand.

Mein besonderer Dank gilt Prof. Dr. Volker Claus, der es mir nicht nur ermöglichte, als wissenschaftlicher Mitarbeiter in seiner Abteilung zu arbeiten und diese Arbeit zu schreiben, sondern darüber hinaus trotz eines immens gefüllten Terminkalenders immer ein offenes und verständnisvolles Ohr hatte, sowohl als fachlicher wie auch als persönlicher Ratgeber.

Prof. Dr. Ulrich Hertrampf danke ich für die Übernahme des Mitberichts.

Zusammenfassung

Die Kürzeste-Wege-Suche ist eines der meistbearbeiteten Gebiete der Graphentheorie – tausende Artikel wurden auf Konferenzen vorgestellt oder in Zeitschriften veröffentlicht. Die Übersichtsarbeiten von Deo und Pang ([DP84]) und Cherkassky et. al. ([CGR96]) sowie Zwick ([Zwi01]) geben einen guten Überblick über die Vielfalt der Problemstellungen und Ansätze zur Lösung Kürzester-Wege-Probleme. Diese Arbeit behandelt darüber hinaus sehr aktuelle Algorithmen, die erst in den letzten zwei Jahren entwickelt und veröffentlicht wurden.

Wir geben in dieser Arbeit einen anderen Blickwinkel auf diese Vielfalt der Algorithmen – bei der Herleitung der Algorithmen wird der Schwerpunkt auf die Herausarbeitung von Gemeinsamkeiten und Unterschieden gelegt. Dabei achten wir auf eine logische, nicht notwendigerweise chronologische, Reihenfolge. Um sich nicht in der Vielfalt der Problemstellungen zu verlieren, beschäftigt sich diese Arbeit im Wesentlichen mit der Suche der kürzesten Wege von *einem* Startknoten zu allen anderen Knoten – andere Problemstellungen werden nur kurz behandelt.

Ausgehend von grundlegenden Eigenschaften leiten wir einen generischen Kürzeste-Wege-Algorithmus her, der fast alle geläufigen Kürzeste-Wege-Algorithmen als Sonderfälle umfasst – einige davon werden hier auch im Detail betrachtet. Darüber hinaus werden wir verschiedene Techniken und Datenstrukturen vorstellen, die die Suche beschleunigen können. Dabei werden an vielen Stellen durch kommentierende Bemerkungen die Aussagen der Originalarbeiten vertieft und im Hinblick auf Gemeinsamkeiten mit anderen Algorithmen erweitert.

Neben diesen strukturellen Ergebnissen liefert diese Arbeit neue Aussagen insbesondere zur beidseitigen heuristischen Suche, zur Verwendung von Bucketstrukturen bei reellen Kantengewichten und eine Verbesserung eines k-kürzeste-Wege-Algorithmus.

Abstract

The shortest path problem is one of the most studied in graph theory. Thousands of papers have been published at conferences or in journals. The surveys by Deo and Pang ([DP84]), Cherkassky et. al. ([CGR96]), as well as Zwick ([Zwi01]) provide an insight into the variety of questions and solutions in shortest path computation. This dissertation also considers some algorithms that were recently published within the past two years.

We give a different point of view to the variety of algorithms for shortest path computation – the main focus is on the similarities of these algorithms and how they evolve from each other. We use a logical order rather than a chronological one. It's easy to get lost in the diversity of shortest path problems, so we focus on the computation of shortest paths from *one* node to all the others. Other problems are briefly described.

Beginning with basic properties of shortest paths we deduce a generic shortest path algorithm. Almost all well-known shortest path algorithms are special cases of this generic one. Some of these will be described in detail. Furthermore different techniques and data structures will be shown to be efficient in shortest path computation. We give many additional comments that extend the original papers, so as to show similarities between the algorithms.

In addition to these structural results, we will provide new theorems. The main results: Concerning bi-directional heuristic search we show that an unidirectional heuristic search is always faster than the bi-directional one. In contrast to this, the non-heuristic bi-directional search is almost always faster than an unidirectional one. We show how to use bucket structures in real weighted graphs by combining an old theorem by Dinic ([Din78]) with algorithms for integer weighted graphs. We give an improved algorithm for computing the k shortest paths.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Notationen und Begriffe

$\mathbb{N}, \mathbb{N}_0$	Menge der natürlichen Zahlen (einschließlich der 0)
$\mathbb{R}, \mathbb{R}^+, \mathbb{R}_0^+$	Menge der (echt positiven, nichtnegativen) reellen Zahlen
$\mathrm{d}^{(e)}(p_1, p_2)$	euklidischer Abstand zwischen den Punkten $p_1, p_2 \in \mathbb{R}^2$
U, V	Knotenmengen
E	Kantenmenge
$G = (V, E, \ldots)$	Graph (ggf. mit weiteren Attributen)
$T = (V, E_T)$	Spannbaum des Graphen $G = (V, E)$
n, m	Kardinalität von V bzw. E
u, v, x, y	Knoten eines Graphen
uv	ungerichtete Kante eines Graphen, auch für die gerichtete Kante (u, v)
s, v_0, z, v_k	Start-/Zielknoten einer Kürzeste-Wege-Suche
$w, u \overset{w}{\rightsquigarrow} v$	ein Weg w (von u nach v)
APSP-Problem	All-Pairs-Shortest-Path-Problem
SSSP-Problem	Single-Source-Shortest-Path-Problem
SPSP-Problem	Single-Pair-Shortest-Path-Problem
$\gamma(e)$	Länge der Kante e
$\gamma_{\min}$, $\gamma_{\max}$, $\gamma_{\lvert\max\rvert}$	kleinstes, größtes, betragsmäßig größtes Kantengewicht
$\gamma(w)$, $\lvert w\rvert$	Länge des Weges w, Anzahl der Kanten in w
G_π, $\gamma_\pi(\cdot)$	(gewichts-) reduzierter Graph, reduzierte Kantengewichte
$\mathrm{d}_s(v)$, $\mathrm{dist}(s, v)$	kürzeste Entfernung von s nach v
$\mathrm{D}_s(v)$	berechnete obere Schranke von $\mathrm{d}_s(v)$
$\widehat{\mathrm{ed}}_z(v)$	geschätzte Entfernung von Knoten v nach z
B_s	Menge der Knoten v, für die $\mathrm{d}_s(v)$ berechnet wurde
$N^i(U)$	Menge der Knoten v, die über höchstens i Kanten von einem Knoten aus U aus erreichbar sind
R_s	Menge der Knoten $N^1(B_s) - B_s$, für die eine Kante (u, v), $u \in B_s$, $v \notin B_s$ existiert
i, j, k	Indizes
$\alpha(m, n)$	Inverse der Ackermann-Funktion

Kapitel 1

Einleitung

Die Kürzeste-Wege-Suche ist eines der klassischen Probleme der Graphentheorie. Es lässt sich leicht beschreiben und mit einfachen Polynomialzeit-Algorithmen lösen. Seit den grundlegenden Algorithmen von Bellman und Ford ([Bel58], [For56]) sowie Dijkstra ([Dij59]) sind nun fast fünfzig Jahre vergangen, in denen sich tausende Veröffentlichungen mit der Kürzeste-Wege-Suche in allen Variationen beschäftigt haben. Nichtsdestotrotz sind selbst für die klassischen Probleme der Kürzeste-Wege-Suche – z. B. von einem Startknoten zu allen anderen – noch keine (beweisbar) optimalen Algorithmen bekannt. Diese existieren bislang nur für spezielle Graphenklassen (z. B. ein Linearzeit-Algorithmus für ungerichtete Graphen mit ganzzahligen positiven Kantengewichten von Thorup ([Tho99]) oder ein Linearzeit-Algorithmus für planare Graphen mit positiven Kantengewichten ([HKRS97])). Für die meisten Graphenklassen und Kürzeste-Wege-Probleme gibt es jedoch zwischen den theoretischen unteren Schranken und den bekannten Algorithmen noch kleinere oder größere Lücken. Auch spielt das Berechnungsmodell für die Laufzeiten eine entscheidene Rolle: Der oben genannte Linearzeit-Algorithmus nach [Tho99] benötigt z. B. ein RAM-Modell, in dem AC^0-Operationen in konstanter Zeit durchgeführt werden können. Dieses ist zwar im RAM-Modell durchaus üblich, betrachtet man aber denselben Algorithmus auf einem vergleichsbasierten Modell, das nur einfache Operationen wie Additionen und Vergleiche in konstanter Zeit ausführen kann, so benötigt dieser im schlechtesten Fall mindestens $\Omega(m + n \log n)$ Schritte ([Pet04]). Neuere Algorithmen versuchen im Mittel optimal zu sein, so lösen z. B. die Algorithmen von Goldberg ([Gol04]) und Hagerup ([Hag04]) das Kürzeste-Wege-Problem im Mittel in Linearzeit (der Worst-Case ist jedoch nicht linear).

Im täglichen Leben begegnen uns Kürzeste-Wege-Suchen meist in der Form „Suche von einem Start- zu einem Zielknoten“ (Navigationssysteme im Auto oder Fahrplanauskünfte im öffentlichen Nahverkehr). Wir werden uns in dieser Arbeit hauptsächlich mit diesem Problem – Kürzeste-Wege-Suche von *einem* Startknoten aus – beschäftigen, dabei aber die kürzesten Wege zu *allen* Zielknoten berechnen. Andere Problemstellungen, wie die Berechnung zweitkürzester Wege oder die Berechnung der kürzesten Wege zwischen allen Paaren von Start- und Zielknoten, werden wir nur am Rande betrachten – für weitere zumindest noch im Ausblick Literaturhinweise geben.

Zum Aufbau der Arbeit

Der Fokus dieser Arbeit liegt in einer einheitlichen Darstellung und Klassifizierung Kürzester-Wege-Algorithmen. Dabei werden die Algorithmen logisch aufeinander aufbauend hergeleitet und so Gemeinsamkeiten verdeutlicht, auch wenn dabei die chronologische Reihenfolge nicht immer eingehalten wird.

Neue Ergebnisse dieser Arbeit, z. B. im Rahmen der beidseitigen Suche, sind deshalb in die Herleitung der Algorithmen eingebettet und nicht in einem separaten Kapitel dargestellt.

Das Verständnis der Originalarbeiten wird durch kommentierende Bemerkungen vertieft. Hierbei wird auf die Darstellung der (nicht immer offensichtlichen) Gemeinsamkeiten und Gegensätzlichkeiten verschiedener Algorithmen Wert gelegt. Häufig werden auch Verbesserungen oder Erweiterungen der Aussagen der Originalartikel hergeleitet oder diese aus einem anderern Blickwinkel heraus betrachtet. So sind auch fast alle Beweise und Darstellungen gegenüber den Originalarbeiten neu verfasst, um so die Gemeinsamkeiten zu anderen Algorithmen besser darstellen zu können.

In Kapitel 2 werden die grundlegenden hier verwendeten Notationen und Begriffe eingeführt. Der mit den Begriffen der Graphentheorie und mit Komplexitätsmaßen vertraute Leser mag diesen Abschnitt überspringen. Weitergehende Definitionen werden an entsprechender Stelle gegeben.

Das Kapitel 3 gibt eine Einführung in Kürzeste-Wege-Algorithmen. Ausgehend von grundlegenden Eigenschaften wird ein generischer Algorithmus entworfen, aus dem die geläufigen Algorithmen, z. B. von Bellman-Ford und Dijkstra, aber auch über eine Grundvorlesung hinausgehende Algorithmen, z. B. von D'Esopo-Pape und Pallottino, hergeleitet werden.

Speziell für die Suche von *einem* Start- zu *einem* Zielknoten sind die in Kapitel 4 vorgestellten Algorithmen zur beidseitigen Suche geeignet. Während

eine heuristische Suche im unidirektionalen Fall nur Verbesserungen bringen kann, werden wir hier eines der wichtigsten Ergebnisse dieser Arbeit zeigen, dass die klassische beidseitige heuristische Suche stets einer der unidirektionalen Varianten unterlegen sein muss. Ebenfalls neu sind Abschätzungen über die Güte von Näherungslösungen, die über beidseitige Suchansätze gewonnen werden.

In Graphen mit ganzzahligen Kantengewichten werden meist Bucket-Strukturen zur Beschleunigung der Berechnung verwendet. Wir werden diese in Kapitel 5 aufeinander aufbauend vorstellen. Neu ist insbesondere, die Ansätze auf reellwertige Kantengewichte zu erweitern, sodass wir über diesen Weg die Laufzeiten der bisher besten bekannten Algorithmen verbessern können.

Das Kapitel 6 stellt einen mit den Bucket-Strukturen verwandten Ansatz vor, der durch Verkleinerung der Kantengewichte die Kürzeste-Wege-Berechnung zu beschleunigen versucht. Erlaubt man negative Kantengewichte, so ist für ganzzahlige Kantengewichte der hier vorgestellte Algorithmus von Goldberg ([Gol95]) für große Bereiche von Kantengewichten dem sonst schnellsten Algorithmus von Bellman-Ford überlegen.

Die folgenden Kapitel beschäftigen sich mit weiteren Ansätzen und Problemstellungen: In Kapitel 7 werden Algorithmen zur Berechnung der kürzesten Wege zwischen allen Knotenpaaren behandelt. Das Kapitel 8 stellt Algorithmen zur Berechnung k-kürzester Wege vor, unter anderem eine Verbesserung eines Algorithmus von [MPS99], sodass dieser nun im vergleichsbasierten Modell optimal ist, wenn der k-kürzeste-Wege-Baum mit ausgegeben werden soll. Das Kapitel 9 bietet noch einen Überblick über Algorithmen in eingeschränkten Graphenklassen (planare Graphen und fast azyklische Graphen), weitere Ansätze im Bereich der Kürzeste-Wege-Suche (z. B. geometrische Beschleunigungsverfahren, die mit den in den ersten Kapiteln vorstellten Algorithmen kombiniert werden können) und einen Überblick über offene Probleme und Ideen, die sich im Laufe dieser Arbeit ergeben haben. Wir schließen mit der Zusammenfassung in Kapitel 10.

Kapitel 2

Definitionen

Dieses Kapitel führt die verwendete Notation und grundlegende Definitionen ein. Weitergehende Definitionen, die nur in einzelnen Kapiteln verwendet werden, werden an entsprechender Stelle gegeben.

2.1 Mathematische Definitionen

$\mathbb{N}$ sind die natürlichen Zahlen und $\mathbb{N}_0$ die natürlichen Zahlen einschließlich der 0. $\mathbb{Z}$ sind die ganzen Zahlen, $\mathbb{R}$ die reellen, $\mathbb{R}^+$ die echt positiven reellen und $\mathbb{R}_0^+$ die nichtnegativen reellen Zahlen.

Die untere und obere Gaußklammer ist für alle $x \in \mathbb{R}$ definiert durch $\lfloor x \rfloor = \max\{z \in \mathbb{Z} \mid z \leq x\}$ und $\lceil x \rceil = \min\{z \in \mathbb{Z} \mid z \geq x\}$.

Wenn keine Basis angegeben ist, bezeichnet $\log x$ stets den Logarithmus von $x \in \mathbb{R}^+$ zur Basis 2. Der Logarithmus von $x \in \mathbb{R}^+$ zur Basis e wird mit $\ln x$ bezeichnet.

Wir schreiben abkürzend $f(M) := \{f(m) \mid m \in M\}$ für alle Funktionen $f : X \to Y$ und alle Mengen $M \subseteq X$.

Definition 2.1 (Heap) *Ein* Heap *mit k Elementen ist ein Array $A[1 \ldots k]$ mit der Eigenschaft $A[i \text{ div } 2] \leq A[i]$, $1 < i \leq k$.*

Ein Heap ermöglicht das Einfügen, Löschen und Verändern von Werten in je $O(\log k)$ Schritten und die Bestimmung des Minimums in $O(1)$ (Anhang A).

Definition 2.2 (Euklidischer Abstand) *Der* Euklidische Abstand $\mathrm{d}^{(e)}(\cdot)$ *ist für zwei Punkte* $(p_x, p_y), (q_x, q_y) \in \mathbb{R}^2$ *definiert durch*

$$\mathrm{d}^{(e)}((p_x, p_y), (q_x, q_y)) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Die Funktion $\alpha(m, n) := \min\{i \geq 1 \mid A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}$ ist extrem langsam wachsend und wird in der Regel als Inverse der Ackermann-Funktion bezeichnet. Die Ackermann-Funktion ist durch $A(0, j) = j + 1,\ j \geq 0$, $A(i, 0) = A(i - 1, 1),\ i \geq 1$ und $A(i, j) = A(i - 1, A(i, j - 1)),\ i, j \geq 1$ definiert.

2.2 Graphen

In diesem Abschnitt werden grundlegende Definitionen zu Graphen gegeben. Erweiterungen zu diesen Definitionen werden an entsprechender Stelle eingeführt.

Definition 2.3 (ungerichteter Graph) *Ein* (schlichter) ungerichteter Graph $G = (V, E)$ *ist ein Paar von disjunkten Mengen* V *(der Menge der* Knoten*) und* $E \subseteq \binom{V}{2}$ *(der Menge der* Kanten*). Ist* $e = \{u, v\} =: uv \in E$*, so heißen* u *und* v adjazent *und* u *(bzw.* v*) und* e inzident.[1] *Wir schreiben stets* $n := |V|$ *und* $m := |E|$.

Definition 2.4 (gerichteter Graph) *Ein* (schlichter) gerichteter Graph $G = (V, E)$ *ist ein Paar von disjunkten Mengen* V *(der Menge der* Knoten*) und* $E \subseteq V \times V - \{(v, v) \mid v \in V\}$ *(der Menge der* Kanten*). Ist* $e = (u, v) =: uv \in E$*, so heißen* u *und* v adjazent *und* u *(bzw.* v*) und* e inzident*. Wir schreiben stets* $n := |V|$ *und* $m := |E|$.

Die folgenden Definitionen und auch alle später betrachteten Algorithmen orientieren sich an gerichteten Graphen. Für ungerichtete Graphen sind sie sinngemäß zu übertragen.

Definition 2.5 (Untergraphen) *Ist* $G = (V, E)$ *und* $U \subseteq V$*, so ist der* von U induzierte Untergraph $G(U) = (U, E')$ *der Graph mit* $E' = (U \times U) \cap E$*. Ist* E' *nur Teilmenge von* $(U \times U) \cap E$*, so heißt* $G' = (U, E')$ Untergraph *von* G.

[1] Die Reihenfolge, in der die Knoten genannt werden, ist bei ungerichteten Graphen beliebig – uv und vu bezeichnen dieselbe Kante.

Definition 2.6 (Wege/Erreichbarkeit) *Ein* Weg w *mit* Startpunkt v_0 *und* Endpunkt v_k *ist eine Folge von Knoten* $w = v_0 v_1 \cdots v_k$*, sodass* $v_{i-1}v_i \in E$ *für* $i \in \{1, \ldots, k\}$*,* v_k *heißt von* v_0 *aus* erreichbar *(wir schreiben* $v_0 \overset{w}{\rightsquigarrow} v_k$*). Sind (bis auf Start/Endpunkt) die* v_i *paarweise verschieden, so heißt der Weg* doppelpunktfrei*. Ist* $v_0 = v_k$ *und* $k \neq 0$*, so heißt der Weg* (doppelpunktfreier) Kreis *oder* Zyklus*.*

Definition 2.7 (Zusammenhang) *Existiert im Graphen* $G = (V, E)$ *für alle* $u, v \in V$*,* $u \neq v$ *ein Weg von* u *nach* v*, so heißt* G (stark) zusammenhängend*. Der Graph* $G(U)$ *heißt* (starke) Zusammenhangskomponente von G *genau dann, wenn* $G(U)$ *(stark) zusammenhängend und* U *maximal bzgl. Inklusion ist.*[2] *Ein gerichteter Graph* $G = (V, E)$ *heißt* schwach zusammenhängend *genau dann, wenn der ungerichtete Graph* $G' = (V, E')$*,* $E' := \{uv \mid uv \in E \ \vee \ vu \in E\}$ *zusammenhängend ist. Der Graph* $G(U)$ *heißt* schwache Zusammenhangskomponente von G *genau dann, wenn* $G'(U)$ *Zusammenhangskomponente von* G' *ist.*

Definition 2.8 (Nachbarschaftsrelation) *Die* Nachbarschaftsrelation *für Knotenmengen* $U \subseteq V$ *ist über die Anzahl der Kanten auf verbindenden Wegen definiert:*

$$\begin{aligned} N^0(U) &:= U \\ N^{i+1}(U) &:= N^i(U) \cup \{u' \mid \exists uu' \in E \text{ mit } u \in N^i(U)\} \\ N^*(U) &:= N^{|V|-1}(U) \end{aligned}$$

Wir schreiben abkürzend $N^i(v)$ *statt* $N^i(\{v\})$*.*

$N^i(v)$ enthält genau die Knoten $u \in V$, die über maximal i Kanten von v aus erreichbar sind. In ungerichteten Graphen ist $N^*(v)$ die Zusammenhangskomponente, in der v liegt.

Definition 2.9 ((Spann-)Baum) *Ein gerichteter Graph* $G = (V, E)$ *heißt* Baum *mit* Wurzel $v_0 \in V$*, wenn für alle* $v \in V$ *genau ein Weg* $v_0 \cdots v$ *in* G *existiert. Ein* Spannbaum *eines (zusammenhängenden) Graphen* $G = (V, E)$ *ist ein Baum* $T = (V, E_T)$ *mit* $E_T \subseteq E$*.*[3]

[2] Bei ungerichteten Graphen lässt man das *stark* weg und sagt nur *zusammenhängend* bzw. *Zusammenhangskomponente*.

[3] Im ungerichteten Baum kann jeder Knoten die Funktion der Wurzel übernehmen. Ein ungerichteter Graph ist somit ein Baum, wenn er zusammenhängend und zyklenfrei ist.

Definition 2.10 (planar) *Existiert zu einem Graphen $G = (V, E)$ eine Darstellung der Knoten $v \in V$ durch Punkte $p_v \in \mathbb{R}^2$ und der Kanten $uv \in E$ durch Linien l_{uv}, die p_u und p_v verbinden, sodass $u \neq v \Rightarrow p_u \neq p_v$ für alle $u, v \in V$ und keine zwei Linien l_{uv}, $l_{u'v'}$ einander kreuzen, so heißt G* planar.

Weitere Definitionen werden an entsprechender Stelle gegeben.

2.3 Gewichtete Graphen

Definition 2.11 (gewichteter Graph) *Sei $\gamma : E \to \mathbb{R}$ eine totale Abbildung, so heißt $G = (V, E, \gamma)$* gewichteter Graph. *Es seien $\gamma_{\min} := \min(\gamma(E))$, $\gamma_{\max} := \max(\gamma(E))$ und $\gamma_{|\max|} := \max\{|\gamma_{\min}|, |\gamma_{\max}|\}$.*

Definition 2.12 (Euklidischer Graph) *Sei $G = (V, E, \gamma)$ ein gewichteter Graph und $\kappa : V \to \mathbb{R}^2$ eine Einbettung in die Ebene, sodass $\gamma(uv) = \mathrm{d}^{(e)}(u, v)$, $uv \in E$, so heißt $G = (V, E, \gamma)$ mit der Einbettung κ* Euklidischer Graph.

Definition 2.13 (Weglängen / Entfernungsfunktion) *Im gewichteten Graphen $G = (V, E, \gamma)$ wird γ auf Wege $w = v_0 v_1 \dots v_k$ fortgesetzt mit $\gamma(w) = \sum_{i=1}^{k} \gamma(v_{i-1} v_i)$. Die* Entfernungsfunktion

$$\mathrm{dist}(v_0, v_k) := \min\{\gamma(w) \mid w \text{ ist Weg von } v_0 \text{ nach } v_k\}$$

gibt die Länge der kürzesten Wege an (ggf. ist $\mathrm{dist}(v_0, v_k) = \infty$, falls kein Weg existiert, oder $\mathrm{dist}(v_0, v_k) = -\infty$, falls ein Weg von v_0 nach v_k existiert, der einen Zyklus negativer Länge als Teilweg enthält). Mit $|w|$ bezeichnen wir die Anzahl der Kanten in w.

Wir fassen im Weiteren Kanten als Wege auf und schreiben stets $\gamma(uv)$ statt $\gamma((u, v))$ und $\gamma(\{u, v\})$.

Wir betrachten auch in gewichteten Graphen keine Schlingen (Kanten von einem Knoten auf sich selbst), da sich durch diese die Weglängen nicht verkürzen können bzw. bei Schlingen negativer Länge diese beliebig oft durchlaufen werden können und es so keinen kürzesten Weg geben kann. Ebenso brauchen wir Multikanten (mehrere Kanten, die dieselben Knoten u und v verbinden) nicht zu berücksichtigen und beschränken uns ggf. auf die jeweils kürzeste dieser Kanten.

Definition 2.14 (minimaler Spannbaum) *Ein* minimaler Spannbaum *eines (zusammenhängenden) Graphen $G = (V, E, \gamma)$ ist ein Spannbaum $T = (V, E_T)$ mit $\sum_{e \in E_T} \gamma(e) \leq \sum_{e \in E_{T'}} \gamma(e)$ für alle Spannbäume $T' = (V, E_{T'})$ von G.*

2.4 Komplexitätsmaße

Alle Komplexitätsabschätzungen werden im uniformen Komplexitätsmaß angegeben. Wir arbeiten dabei auf einer RAM (Random Access Machine) mit folgenden Konventionen ([AHU75],[CR73],[Hag98]):

- Jede Speicherzelle kann beliebig große Werte speichern.[4] Reelle Zahlen werden mit beliebiger Genauigkeit bearbeitet.
- Mathematische Standardoperationen wie Addition, Vergleiche usw. werden in konstanter Zeit durchgeführt. Die auftretenden Divisionen und Modulo-Rechnungen sind in der Regel dergestalt, dass sie durch Bitoperationen durchgeführt werden können, die Schaltkreisen konstanter Tiefe entsprechen (Schaltkreiskomplexität AC^0). Auch diese haben hier Aufwand $O(1)$.
- Im erweiterten RAM-Modell können auch Multiplikationen und Divisionen sowie andere Funktion außerhalb AC^0 in konstanter Zeit durchgeführt werden. Wir werden diese Operationen nur in wenigen Ausnahmen verwenden.

Beschränkt man sich auf die Operationen Addition und Vergleich (auch keine Indexberechnungen), so spricht man von einem vergleichsbasierten Modell. Alle Algorithmen in den Kapiteln 3, 4 und 8 haben diese Eigenschaft.

Die Abschätzungen werden in Abhängigkeit der Anzahl der Knoten und Kanten im Graphen oder ggf. in Abhängigkeit weiterer Eigenschaften des Graphen angegeben. Wir verwenden dabei die O-Notation. Es gilt für Funktionen $f, g : \mathbb{N}_0 \to \mathbb{R}_0^+$:

- $f \in O(g) \ :\Leftrightarrow \ \exists c \in \mathbb{R}^+,\ n_0 \in \mathbb{N} :\ f(n) \leq c \cdot g(n),\ n \geq n_0$,
  d. h., f wächst asymptotisch nicht schneller als g

[4] Im Allgemeinen fordert man bei ganzzahligen Werten, dass die Anzahl der Bits je Speicherzelle durch $O(\log n + \log \gamma_{|\max|})$ beschränkt ist, sodass die Anzahl der Knoten und Kanten bzw. die auftretenden Gewichte jeweils in einer Speicherzelle abgelegt werden können, es aber nicht möglich ist, beliebig viele Werte in eine Zahl zu kodieren.

- $f \in o(g) \;:\Leftrightarrow\; \forall c \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N} :\ f(n) \leq c \cdot g(n),\ n \geq n_0$,
  d. h., f wächst asymptotisch echt langsamer als g
- $f \in \Omega(g) \;:\Leftrightarrow\; g \in O(f)$,
  d. h., f wächst asymptotisch nicht langsamer als g
- $f \in \omega(g) \;:\Leftrightarrow\; g \in o(f)$,
  d. h., f wächst asymptotisch echt schneller als g
- $f \in \Theta(g) \;:\Leftrightarrow\; f \in O(g) \wedge g \in O(f)$,
  d. h., f wächst asymptotisch gleich schnell wie g

An manchen Stellen verwenden wir auch die $\tilde{O}$-Notation [CLRS01]. Dabei werden im Vergleich zur O-Notation polylogarithmische Faktoren ignoriert.

- $f \in \tilde{O}(g) \;:\Leftrightarrow\; \exists c \in \mathbb{R}^+,\ n_0, k \in \mathbb{N} :\ f(n) \leq c \cdot g(n) \cdot \log^k(n),\ n \geq n_0$,
  d. h., f wächst bis auf polylogarithmische Faktoren asymptotisch nicht schneller als g.

Bemerkung: Es gilt nicht

$$f \in \tilde{O}(g) \;\Leftarrow\; \forall \varepsilon > 0 :\ f \in O(g \cdot n^{\varepsilon})$$

Für die Funktion $f(n) = n^{1/\log\log n}$ gilt z. B. $f(n) \notin \tilde{O}(1)$, da für alle c

$$\begin{aligned} f(n) = 2^{\log(n^{1/\log\log n})} &= 2^{(\log n/(\log\log n)^2)\cdot\log\log n} \\ &= (\log n)^{(\log n/(\log\log n)^2)} \in \omega(\log^c n) \end{aligned}$$

Jedoch ist $f(n)$ in $o(n^{\varepsilon})$ für alle ε. Andererseits gilt nicht für jede Funktion $f(n) = n^{o(1)}$, dass $f(n) \notin \tilde{O}(1)$, z. B. ist $f(n) = n^{1/\log(n)} = 2 \in \tilde{O}(1)$.

Kapitel 3

Einführung in Kürzeste-Wege-Algorithmen

Nach einer kurzen Einführung und Übersicht über die Laufzeiten der derzeit schnellsten bekannten Verfahren wird aus elementaren Eigenschaften kürzester Wege ein generischer Kürzeste-Wege-Algorithmus hergeleitet, der uns als Grundlage für die Beschreibung und Verifikation der Algorithmen in den Folgekapiteln dienen wird.

Man unterscheidet drei Klassen von Kürzeste-Wege-Problemen:

1. SPSP (single-pair shortest-path): Gegeben ist ein Graph $G = (V, E, \gamma)$ und zwei Knoten $s, z \in V$. Gesucht ist ein kürzester Weg von s nach z.

2. SSSP (single-source shortest-path): Gegeben ist ein Graph $G = (V, E, \gamma)$ und ein Startknoten $s \in V$. Gesucht sind für alle Knoten $v \in V - \{s\}$ kürzeste Wege von s nach v. Analog sucht man beim SDSP-Problem (single-destination shortest-path) zu einem Zielknoten $z \in V$ für alle Knoten $v \in V - \{z\}$ kürzeste Wege von v nach z.

3. APSP (all-pairs shortest-path): Gegeben ist ein Graph $G = (V, E, \gamma)$. Gesucht sind für alle Knoten $u, v \in V$ kürzeste Wege von u nach v.

Die Algorithmen berechnen dabei die Längen der kürzesten Wege und bauen eine Datenstruktur auf, die es erlaubt, für die berechneten Knotenpaare $v_i, v_j \in V$ einen kürzesten Weg w in $O(|w|)$ auszugeben. Existieren mehrere kürzeste Wege zwischen v_i und v_j, so wird nur einer dieser kürzesten Wege berechnet und ausgegeben. Sollen nur die Längen der kürzesten Wege berechnet werden, so spricht man vom SPD- (single pair distance), SSD-

(single source distance) und APD-Problem (all pairs distance). Asymptotisch können jedoch für das SPD- und SSD-Problem keine schnelleren Algorithmen als für die SP-Varianten existieren (vgl. Lemma 3.7). [KKP93] zeigen für Algorithmen, die die Lösung eines APD-Problems verifizieren und dabei auf die Kantengewichte nur zugreifen, indem Pfadlängen verglichen werden, eine untere Schranke für die Laufzeit von $\Omega(nm)$. Damit ist auch das eigentliche APD-Problem mit dieser Einschränkung in $\Omega(nm)$.[1] Das Lemma 3.7 erlaubt die Berechnung der kürzesten Wege aus den kürzesten Entfernungen für alle Knotenpaare in $O(nm)$, sodass auch für solche Algorithmen das APD-Problem nicht schneller als das APSP-Problem zu lösen ist.

Die meisten hier vorgestellten Algorithmen lösen das SSSP-Problem. In der Praxis, z. B. bei Navigationssystemen, ist meist nur das SPSP-Problem zu lösen, für dieses ist jedoch kein asymptotisch schnellerer Algorithmus bekannt. Mit Ausnahme der beidseitigen Wegealgorithmen sind die Ansätze zur Lösung des SPSP-Problems lediglich Algorithmen zur Lösung des SSSP-Problems, die frühestmöglich abgebrochen werden, und selbst bei der beidseitigen Suche werden zumindest zum Teil SSSP/SDSP-Probleme für den Start- bzw. Zielknoten gelöst. Algorithmen zur Lösung des APSP-Problems werden hier nur kurz im Kapitel 7 betrachtet.

Die (im asymptotischen Sinne) schnellsten Algorithmen zur Lösung des SSSP-Problems sind in der Tabelle 3.1 aufgeführt: Dabei nutzen die Algorithmen nach [Tho99] und [Tho03] die Möglichkeiten einer RAM mit AC^0-Operationen aus. [Gol95] benutzt lediglich Shiftoperationen um je ein Bit. Die anderen Algorithmen arbeiten vergleichsbasiert.

$G = (V, E)$, $\gamma : E \to \cdot$	Laufzeit
G gerichtet, $\gamma : E \to \mathbb{R}$	[Bel58],[For56] $O(nm)$
G gerichtet, $\gamma : E \to \mathbb{Z}$	[Gol95]: $O(\sqrt{n}m \log(-\gamma_{\min}))$
G gerichtet, $\gamma : E \to \mathbb{R}^+$	[Dij59],[FT87]: $O(m + n \log n)$ [PR02]: $O(m + n \log(\gamma_{max}/\gamma_{\min}))$
G gerichtet, $\gamma : E \to \mathbb{N}$	[Tho03]: $O(m + n \log \log n)$, $O(m + n \log \log \gamma_{\max})$
G unger., $\gamma : E \to \mathbb{R}^+$	[PR02]: $O(m\alpha(m, n) + n \log \log(\gamma_{max}/\gamma_{\min}))$
G unger., $\gamma : E \to \mathbb{N}$	[Tho99]: $O(m)$

Tabelle 3.1: Übersicht zu SSSP-Algorithmen

Wir leiten nun aus elementaren Eigenschaften kürzester Wege ein allgemeines

[1] Ansätze zur Lösung des APD/APSP-Problems über Matrixmultiplikation fallen z. B. nicht in diese Algorithmen-Klasse.

Berechnungsschema für kürzeste Wege her, welches die bekannten Verfahren nach Bellman-Ford ([Bel58], [For56]), D'Esopo-Pape ([PW60], [Pap74], [Pap80], [Pap83]), Pallottino ([Pal84]), effiziente Algorithmen für azyklische Graphen und Graphen mit Kantengewichten $\gamma(\cdot) \equiv c,\ c \in \mathbb{R}$, sowie Dijkstra [Dij59] und den A^*-Algorithmus [HNR68] als Sonderfälle umfasst. Dabei werden die Zusammenhänge dieser Algorithmen verdeutlicht. Die meisten grundlegenden Aussagen sind in [AMO93] und [GP86] zu finden, sind dort jedoch nach anderen Kriterien angeordnet. Die Aussagen wurden zum Teil erweitert und die Beweise – wenn nicht explizit mit Quelle angegeben – neu verfasst, um später die Eigenschaften und Gemeinsamkeiten zwischen den Algorithmen besser darstellen zu können.

Fast alle Algorithmen zur Bestimmung der kürzesten Entfernungen basieren auf folgender zentraler Eigenschaft kürzester Wege.

Lemma 3.1 (Teilwege kürzester Wege sind kürzeste Wege) *Sei $G = (V, E, \gamma)$ ein gewichteter gerichteter Graph und $w = v_0 v_1 \cdots v_k$ ein kürzester Weg von v_0 nach v_k, so gilt: $\forall i, j \in \{0, \ldots, k\},\ i < j$, ist $w_{ij} := v_i v_{i+1} \cdots v_j$ ein kürzester Weg von v_i nach v_j.*

Beweis: Angenommen, es gibt einen Weg $w'_{ij} = v_i \cdots v_j$ mit $\gamma(w'_{ij}) < \gamma(w_{ij})$, so wäre

$$w' = v_0 \overset{w_{0i}}{\rightsquigarrow} v_i \overset{w'_{ij}}{\rightsquigarrow} v_j \overset{w_{jk}}{\rightsquigarrow} v_k$$

ein Weg mit $\gamma(w') = \gamma(w) - \gamma(w_{ij}) + \gamma(w'_{ij}) < \gamma(w)$ im Widerspruch zur Annahme, w sei ein kürzester Weg von v_0 nach v_k. □

Hat ein Graph Zyklen negativer Länge[2], so sind kürzeste Wege nicht für alle Knotenpaare (u, v) wohldefiniert. Analog gilt dies in ungerichteten Graphen mit negativen Kantengewichten, da ein negativer Zyklus bzw. im ungerichteten Fall eine Kante uv mit $\gamma(uv) < 0$ beliebig oft durchlaufen werden kann.[3] Beschränkt man sich auf doppelpunktfreie Wege, so zeigt die Abbildung 3.1, dass das Lemma 3.1 dann im Allgemeinen nicht gilt – in ungerichteten Graphen müssen dazu nicht einmal (einfache) Zyklen negativer Länger existieren:

[2]Wir schreiben im Folgenden zur flüssigeren Lesbarkeit „negativer Zyklus".

[3]Wir kommen im Rahmen von Algorithmen für das APSP-Problem, Kapitel 7, darauf zurück, wie man für solche Graphen die kürzesten Entfernungen $\mathrm{dist}(\cdot, \cdot)$ berechnet, wobei dann auch die Werte $-\infty$ oder $+\infty$ ausgegeben werden, falls auf dem Weg ein negativer Zyklus liegen kann bzw. gar kein Weg existiert.

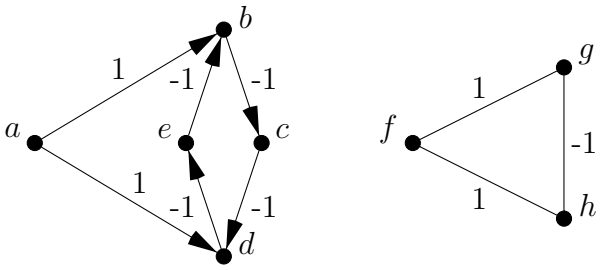


Abbildung 3.1: Kürzeste Wege bei negativen Kantengewichten

Der Weg fgh ist kürzester Weg von f nach h, der kürzeste Weg von f nach g ist jedoch nicht dieser Teilweg, sondern führt über h.[4]

Bei ungerichteten Graphen mit nichtnegativer Gewichtsfunktion und bei gerichteten Graphen ohne negative Zyklen existiert zu jedem Weg w offensichtlich ein doppelpunktfreier Weg w' mit $\gamma(w') \leq \gamma(w)$. Wir berechnen im Folgenden stets die kürzesten Wege ohne Zyklen der Länge 0.

Das Lemma 3.1 bleibt auch in Graphen mit negativen Zyklen gültig, da ein kürzester Weg von v_0 nach v_k nur existiert, wenn kein Weg von v_0 über einen negativen Zyklus zu v_k führt. Die Prämisse der Implikation bleibt somit ggf. stets falsch und die Aussage des Lemmas damit auch in diesem Fall wahr.

3.1 Potentialfunktionen und Kantengewichtstransformationen

Eine Technik, die Wegeprobleme vereinfachen und Zusammenhänge zwischen verschiedenen Algorithmen aufzeigen kann, ist die Transformation von Kantengewichten mit Hilfe von Potentialfunktionen. Ursprünglich bei Flussproblemen genannt ([FF62]), werden sie in [EK72] bei Wegeproblemen erwähnt und finden z. B. beim APSP-Problem ([Joh77]), bei Skalierungsalgorithmen ([Gab85], [GT89], [OA92], [Gol95]) und der beidseitigen Wegesuche ([IHI94]) Anwendung. Wir werden diese in den Kapiteln 7, 6 und 4 ausführen, zeigen hier aber zunächst nur die elementaren Eigenschaften solcher Kantengewichtstransformationen.

[4]Das NP-harte Problem der Berechnung des längsten doppelpunktfreien Weges lässt sich auf die Berechnung des kürzesten doppelpunktfreien Weges in Graphen, in denen auch negative Kantengewichte erlaubt sind, reduzieren. Somit ist auch die Berechnung des kürzesten doppelpunktfreien Weges NP-hart ([GJ79]).

Definition 3.2 (Potentialfunktion, reduzierte Kantengewichte) *Eine Funktion $\pi : V \to \mathbb{R}$, die den Knoten reellwertige Zahlen zuordnet, heißt* Potentialfunktion.

Zum gerichteten Graphen $G = (V, E, \gamma)$ heißt $G_\pi = (V, E, \gamma_\pi)$ der gewichtsreduzierte Graph[5] *mit den* reduzierten Kantengewichten $\gamma_\pi(uv) = \pi(u) + \gamma(uv) - \pi(v)$.

Für ungerichtete Graphen sind der reduzierte Graph bzw. reduzierte Kantengewichte nicht definiert.

Lemma 3.3 (Invarianz bezüglich Zyklen negativer Länge) *Seien $G = (V, E, \gamma)$, $\gamma : E \to \mathbb{R}$ und $\pi : V \to \mathbb{R}$ eine beliebige Potentialfunktion, dann gilt im reduzierten Graphen $G_\pi = (V, E, \gamma_\pi)$ für jeden Zyklus $w = v_0 v_1 \cdots v_k$, $v_0 = v_k$, dass $\gamma(w) = \gamma_\pi(w)$.*

Beweis: Es gilt:

$$\begin{aligned}\gamma_\pi(w) = \sum_{i=1}^{k} \gamma_\pi(v_{i-1}v_i) &= \sum_{i=1}^{k} (\pi(v_{i-1}) + \gamma(v_{i-1}v_i) - \pi(v_i)) \\ &= \pi(v_0) - \pi(v_k) + \sum_{i=1}^{k} \gamma(v_{i-1}v_i) = \gamma(w)\end{aligned}$$

da $\pi(v_0) = \pi(v_k)$ wegen $v_0 = v_k$. □

Insbesondere hat der reduzierte Graph G_π genau dann Zyklen negativer Länge, wenn der Graph G diese hat. Die Wohldefiniertheit kürzester Wege kann also durch die Einführung einer beliebigen Potentialfunktion π nicht zerstört werden.

Die Weglängeneigenschaft für Zyklen überträgt sich in leicht abgewandelter Form auch auf nicht geschlossene Wege. Sei $w = v_0 v_1 \cdots v_k$ ein beliebiger Weg, so gilt:

$$\gamma_\pi(w) = \sum_{i=1}^{k} (\pi(v_{i-1}) + \gamma(v_{i-1}v_i) - \pi(v_i)) = \gamma(w) + \pi(v_0) - \pi(v_k) \qquad (3.4)$$

Wir folgern daraus das Lemma 3.5.

[5] Wir schreiben kurz „reduzierter Graph“. Eine Verwechslungsgefahr mit dem auch so bezeichneten Graphen, der entsteht, wenn man die starken Zusammenhangskomponenten auf jeweils einen Knoten zusammenzieht, ist hier nicht gegeben.

Lemma 3.5 (Invarianz bzgl. der Ordnung der Weglängen) *Seien $G = (V, E, \gamma)$, $\gamma : E \to \mathbb{R}$, $\pi : V \to \mathbb{R}$ eine beliebige Potentialfunktion und $G_\pi = (V, E, \gamma_\pi)$ der reduzierte Graph. Seien $w = v_0 \cdots v_k$ und $w' = v_0 \cdots v_k$ zwei verschiedene Wege von v_0 nach v_k, so gilt $\gamma(w) < \gamma(w')$ genau dann, wenn $\gamma_\pi(w) < \gamma_\pi(w')$. Insbesondere ist $w = v_0 v_1 \cdots v_k$ genau dann kürzester Weg in G, wenn w kürzester Weg in G_π ist.*

Beweis: Gilt $\gamma(w) < \gamma(w')$, so mit Gleichung 3.4 auch

$$\gamma_\pi(w) = \gamma(w) + \pi(v_0) - \pi(v_k) < \gamma(w') + \pi(v_0) - \pi(v_k) = \gamma_\pi(w')$$

Angenommen $w = v_0 v_1 \cdots v_k$ sei kürzester Weg in G, aber nicht kürzester Weg in G_π, so existiert ein Weg $w' = v_0 v'_1 v'_2 \cdots v'_{k'} v_k$ mit $\gamma_\pi(w') < \gamma_\pi(w)$. Dann gilt aber auch

$$\gamma(w') = \gamma_\pi(w') - (\pi(v_0) - \pi(v_k)) < \gamma_\pi(w) - (\pi(v_0) - \pi(v_k)) = \gamma(w)$$

im Widerspruch zur Minimalität von w in G. Die Umkehrungen zeigt man analog. □

3.2 Ein generischer Algorithmus zur Kürzeste-Wege-Suche

Wir verwenden in allen Algorithmen und Aussagen folgende Notation:

- $\mathrm{d}_{v_0}(v) := \mathrm{dist}(v_0, v)$ – Länge eines kürzesten Weges von v_0 nach v.
- $\mathrm{D}_{v_0}(v)$ – Schätzungen für $\mathrm{d}_{v_0}(v)$, in den Algorithmen gilt stets $\mathrm{D}_{v_0}(v) \geq \mathrm{d}_{v_0}(v)$ und gibt die bisher beste bestimmte obere Schranke für $\mathrm{d}_{v_0}(v)$ an.

Ist im Kontext klar, welches v_0 gemeint ist, wird auf den Index verzichtet.

Zur Vereinfachung der Formulierungen sei $G = (V, E, \gamma)$, $\gamma : E \to \mathbb{R}$ im weiteren Verlauf stets ein gerichteter Graph ohne negative Zyklen. Wir werden bei den Algorithmen darauf eingehen, wie man solche negativen Zyklen erkennen kann, ohne die Algorithmen dafür wesentlich verändern zu müssen. Desweiteren seien stets alle Knoten vom Startknoten v_0 aus erreichbar.

Lemma 3.6 (Kürzeste-Wege-Eigenschaft der $\mathrm{d}(\cdot)$) *Ein Weg $w = v_0 v_1 \cdots v_k$ ist genau dann kürzester Weg von v_0 nach v_k, wenn $\mathrm{d}(v_{i-1}) + \gamma(v_{i-1} v_i) = \mathrm{d}(v_i)$ für alle $i \in \{1, 2, \ldots, k\}$ gilt.*

Beweis: Ist w kürzester Weg von v_0 nach v_k, so ist nach Lemma 3.1 jeder Teilweg $v_0 v_1 \cdots v_j$ kürzester Weg und somit

$$\mathrm{d}(v_i) = \gamma(v_0 \cdots v_i) = \gamma(v_0 \cdots v_{i-1}) + \gamma(v_{i-1}v_i) = \mathrm{d}(v_{i-1}) + \gamma(v_{i-1}v_i)$$

für alle $i \in \{1, 2, \ldots, k\}$. Gilt umgekehrt $\mathrm{d}(v_{i-1}) + \gamma(v_{i-1}v_i) = \mathrm{d}(v_i)$ für alle $i \in \{1, 2, \ldots, k\}$, so ist mit $\mathrm{d}(v_0) = 0$

$$\gamma(w) = \sum_{i=1}^{k} \gamma(v_{i-1}v_i) = \sum_{i=1}^{k} (\mathrm{d}(v_i) - \mathrm{d}(v_{i-1})) = \mathrm{d}(v_k) - \mathrm{d}(v_0) = \mathrm{d}(v_k)$$

und somit w ein kürzester Weg von v_0 nach v_k. □

Insbesondere kann es keine Kante uv mit $\mathrm{d}(u) + \gamma(uv) < \mathrm{d}(v)$ geben, da es sonst über u einen kürzeren Weg nach v als den (vermeintlich) kürzesten Weg geben würde. Es gilt für alle Kanten uv, dass $\mathrm{d}(u) + \gamma(uv) \geq \mathrm{d}(v)$ ist.

Aus Lemma 3.6 folgt, dass es zu einem Graphen $G = (V, E, \gamma)$ und einem Knoten v_0 stets einen Baum $T = (V, E_T, \gamma)$ mit Wurzel v_0 gibt, sodass die eindeutigen Wege in T genau die Länge $\mathrm{d}_{v_0}(\cdot)$ haben. Wir nennen diesen Baum *Kürzeste-Wege-Baum*. Dieser erlaubt für jeden Knoten v einen kürzesten Weg $w = v_0 \cdots v$ in $O(|w|)$ auszugeben, indem vom Knoten v die Kanten rückwärts bis zur Wurzel v_0 betrachtet werden – dies ist eindeutig möglich, da in einem Baum jeder Knoten (bis auf die Wurzel) genau eine eingehende Kante hat. Dies heißt auch, dass die Speicherung dieser Wege nur $O(n)$ Platz benötigt, während die Gesamtanzahl der Kanten in diesen Wegen $O(n^2)$ beträgt.

Lemma 3.7 (Existenz eines Kürzeste-Wege-Baums) *Seien $G = (V, E, \gamma)$ und $v_0 \in V$, so existiert ein Baum $T = (V, E_T, \gamma)$ so, dass sich die Entfernungen $\mathrm{d}(\cdot)$ in G und T nicht unterscheiden. Sind die $\mathrm{d}(\cdot)$ in G bekannt, so lässt sich T in $O(n + m)$ konstruieren.*

Beweis: Aus Lemma 3.6 folgt, dass für alle Kanten uv aller kürzesten Wege mit Startknoten v_0 die Eigenschaft $\mathrm{d}(u) + \gamma(uv) = \mathrm{d}(v)$ gilt. Man suche in $G' = (V, E', \gamma)$, $E' = \{uv \in E \mid \mathrm{d}(u) + \gamma(uv) = \mathrm{d}(v)\}$ z. B. durch Breiten- oder Tiefensuche in $O(n + m)$ ([CLRS01] u.a.) einen Spannbaum mit Wurzel v_0. In dem so gefundenen Spannbaum sind alle Knoten v_k eindeutig über einen Weg erreichbar. Es gilt für die Länge jedes Wegs $w = v_0 v_1 \cdots v_k$

$$\gamma(w) = \sum_{i=1}^{k} \gamma(v_{i-1}v_i) = \sum_{i=1}^{k} (\mathrm{d}(v_i) - \mathrm{d}(v_{i-1})) = \mathrm{d}(v_k) - \mathrm{d}(v_0) = \mathrm{d}(v_k)$$

da $\mathrm{d}(v_0) = 0$. □

Man beachte, dass ein Kürzeste-Wege-Baum immer existiert, obwohl die kürzesten Wege nicht eindeutig sind – somit auch der Kürzeste-Wege-Baum nicht – und in dem Untergraphen G' sogar Zyklen (der Länge 0) vorhanden sein können.

Möchte man nicht nur jeweils einen, sondern alle kürzesten Wege mit Startknoten v_0 bestimmen, so berechne man G' und zähle alle Wege in G' auf. Zyklen der Länge 0 in G' sollten dabei vorher zu einem Knoten zusammengezogen werden. Dieser muss bei der Aufzählung der Wege dann gesondert betrachtet werden. Die beliebig häufige Wiederholung der Zyklen der Länge 0 könnte man dabei z. B. analog zur Schreibweise bei regulären Ausdrücken mit $\cdots (v_i \cdots v_j)^* \cdots$ markieren.

Korollar 3.8 (Kantengewichte des reduz. Kürzeste-Wege-Baums) *Mit der Potentialfunktion $\pi(\cdot) := \mathrm{d}(\cdot)$ sind die Kanten in E' (aus dem Beweis zu Lemma 3.7) genau die Kanten mit reduziertem Kantengewicht $\gamma_\pi(\cdot) = 0$. Für alle Kanten $uv \in E_T$ gilt $\gamma_\pi(uv) = 0$. Andere Kanten haben nur dann reduzierte Kantenlänge null, wenn sie ebenfalls zu kürzesten Wegen führen (vgl. Lemma 3.6).*

Für alle Kanten uv im Kürzeste-Wege-Baum gilt $\mathrm{d}_{v_0}(v) = \mathrm{d}_{v_0}(u) + \gamma(uv)$. Alle restlichen Kanten können nicht zu kürzeren Wegen führen, es gilt $\mathrm{d}_{v_0}(v) \leq \mathrm{d}_{v_0}(u) + \gamma(uv),\ uv \in E$.[6] Sollen die Werte $\mathrm{D}_{v_0}(\cdot)$ (die Schätzungen für $\mathrm{d}_{v_0}(\cdot)$) die kürzesten Entfernungen vom Startknoten v_0 angeben, so muss also für alle Kanten

$$\mathrm{D}_{v_0}(v) \leq \mathrm{D}_{v_0}(u) + \gamma(uv) \tag{3.9}$$

gelten. Eine Kante mit dieser Eigenschaft nennen wir *konsistent* (gilt die Eigenschaft nicht, so nennen wir die Kante *verkürzend*). Die Gleichung 3.9 ist nicht nur notwendig, sondern auch hinreichend für $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$, sofern zusätzlich $\mathrm{D}(v_0) = 0$ und $\mathrm{D}(\cdot) \geq \mathrm{d}(\cdot)$ gilt.

Lemma 3.10 (Kürzeste-Wege-Eigenschaft der $\mathrm{D}(\cdot)$) *Aus $\mathrm{D}(v_0) = 0$, $\mathrm{D}(v) \geq \mathrm{d}(v)$ und $\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma(uv)$ für alle $u, v \in V$, $uv \in E$ folgt $\mathrm{D}(v) = \mathrm{d}(v)$ ([AMO93] u.a.).*

[6] Durch wiederholte Anwendung entlang des kürzesten Weges von u nach v, $u, v \in V$ beliebig, lässt sich daraus leicht die Dreiecksungleichung für kürzeste Wege $\mathrm{dist}(v_0, v) \leq \mathrm{dist}(v_0, u) + \mathrm{dist}(u, v)$ herleiten. Hat ein Graph einen negativen Zyklus $w = u_0 u_1 \cdots u_k$, $u_0 = u_k$, so folgt auch

$$\mathrm{d}(u_0) = \mathrm{d}(u_k) \leq \mathrm{d}(u_{k-1}) + \gamma(u_{k-1}u_k) \leq \cdots \leq \mathrm{d}(u_0) + \gamma(w)$$

im Widerspruch entweder zur Wohldefiniertheit der $\mathrm{d}(\cdot)$ oder zur Existenz des negativen Zyklus w.

Beweis: Sei $v_0 v_1 \dots v_k$ ein beliebiger Weg von v_0 nach v_k, so gilt:

$$\begin{aligned} \mathrm{D}(v_k) &\leq \mathrm{D}(v_{k-1}) + \gamma(v_{k-1}v_k) \\ \mathrm{D}(v_{k-1}) &\leq \mathrm{D}(v_{k-2}) + \gamma(v_{k-2}v_{k-1}) \\ &\vdots \\ \mathrm{D}(v_1) &\leq \mathrm{D}(v_0) + \gamma(v_0 v_1) \end{aligned}$$

Durch Aufsummieren und mit $\mathrm{D}(v_0) = 0$ ist gezeigt, dass $\mathrm{D}(v_k) \leq \gamma(v_0 \cdots v_k)$ eine untere Schranke für die Weglänge jedes Wegs von v_0 nach v_k ist, insbesondere auch für den kürzesten Weg. Mit $\mathrm{D}(v_k) \geq \mathrm{d}(v_k)$ folgt direkt $\mathrm{D}(v_k) = \mathrm{d}(v_k)$. □

Bevor wir mit Hilfe der Gleichung 3.9 und des Lemmas 3.10 ein allgemeines Schema zur Lösung des SSSP-Problems angeben, betrachten wir zunächst im Zusammenhang mit Potentialfunktionen drei bemerkenswerte Eigenschaften.

Lemma 3.11 (Reduzierte Gewichte konsistenter Kanten) *Seien $G = (V, E, \gamma)$ und $\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma(uv)\ \forall\ u, v \in V,\ uv \in E$, so gilt mit der Potentialfunktion $\pi(\cdot) = \mathrm{D}(\cdot)$ in $G_\pi = (V, E, \gamma_\pi)$ für die reduzierten Kantengewichte $\gamma_\pi(uv) \geq 0$ für alle $uv \in E$.*

Beweis: Einsetzen von $\gamma_\pi(uv) = \gamma(uv) + \mathrm{D}(u) - \mathrm{D}(v)$ in 3.9 ergibt $\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma_\pi(uv) - \mathrm{D}(u) + \mathrm{D}(v)$ und somit folgt die Behauptung. □

Wir nennen eine Potentialfunktion $\pi(\cdot)$ mit $\gamma_\pi(uv) \geq 0$ für alle $uv \in E$ konsistent.

Bemerkung 3.12 *Man beachte, dass im Lemma 3.11 nicht unbedingt $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ gelten muss. Mit $\mathrm{D}(v_0) = 0$ kann $\mathrm{D}(\cdot)$ die kürzesten Entfernungen z. B. unterschätzen, die reduzierten Kantengewichte bleiben (konsistentes $\mathrm{D}(\cdot)$ vorausgesetzt) positiv.*

Analog dazu sind die verkürzenden Kanten uv genau diejenigen, die mit $\pi(\cdot) := \mathrm{D}(\cdot)$ negative reduzierte Kantengewichte $\gamma_\pi(uv) < 0$ haben.

Lemma 3.13 (Linearzeittest für $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$) *Seien für $G = (V, E, \gamma)$ Schätzungen $\mathrm{D}(\cdot)$ für die kürzesten Entfernungen $\mathrm{d}(\cdot)$ gegeben, so lässt sich in $O(n + m)$ überprüfen, ob $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ erfüllt ist.*

Beweis: Man betrachte den reduzierten Graphen $G_\pi = (V, E, \gamma_\pi)$ bezüglich $\pi(\cdot) := \mathrm{D}(\cdot)$. Ist die notwendige Konsistenzeigenschaft für alle Kanten erfüllt

(Gleichung 3.9), so sind nach Lemma 3.11 die Kantengewichte $\gamma_\pi(\cdot) \geq 0$. Soll $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ gelten, so muss nach Korollar 3.8 ein Kürzeste-Wege-Baum T_π existieren, der nur aus Kanten uv mit reduziertem Gewicht $\gamma_\pi(uv) = 0$ besteht. Dessen eindeutige Wege $w_k = v_0 \cdots v_k$ haben reduzierte Länge $\gamma_\pi(w_k) = 0$, d. h., nach Gleichung 3.4 gilt $\gamma_\pi(w_k) = 0 = \gamma(w_k) + \mathrm{D}(v_0) - \mathrm{D}(v_k)$, somit $\mathrm{D}(v_k) = \gamma(w_k)$ und $\gamma(w_k) \geq \mathrm{d}(v_k)$. Mit der Konsistenz und Lemma 3.10 folgt $\mathrm{D}(v_k) = \mathrm{d}(v_k)$. Die Existenz von T_π mit $\gamma_\pi(uv) = 0,\ uv \in E_{T_\pi}$ lässt sich durch eine Breitensuche in Linearzeit überprüfen (vgl. Lemma 3.7). □

Lemma 3.14 (Nichtnegative reduzierte Kantengewichte, [Tur96])
Sei $G = (V, E, \gamma)$. Es existiert eine konsistente Potentialfunktion $\pi(\cdot)$, d. h., in $G_\pi = (V, E, \gamma_\pi)$ gilt $\gamma_\pi(uv) \geq 0$ für alle $uv \in E$, genau dann, wenn G keine Zyklen negativer Länge hat.

Beweis: Hat G Zyklen negativer Länge, so nach Lemma 3.3 auch G_π für beliebige Potentialfunktionen $\pi(\cdot)$. Es können also nicht alle Kanten eines negativen Zyklus eine nichtnegative reduzierte Länge haben.

Habe G nun keine Zyklen negativer Länge. Wir fügen einen neuen Knoten v_0 zu G hinzu, verbinden diesen mit allen Knoten mit Kanten der Länge 0 ($G' = (V \cup \{v_0\}, E \cup (\{v_0\} \times V), \gamma'),\ v_0 \notin V$ mit $\gamma'(uv) := \gamma(uv),\ uv \in E$, $\gamma'(v_0 v) := 0,\ v \in V$). Somit sind alle Knoten von v_0 aus erreichbar und G' hat genau dann negative Zyklen, wenn G diese hat. Man berechne die kürzesten Wege vom Knoten v_0 zu allen anderen Knoten und wähle diese Entfernungen als Potentialfunktion $\pi(v) := \mathrm{d}_{v_0}(v)$. In G'_π gilt mit Gleichung 3.9 nach Lemma 3.11 $\gamma'_\pi(\cdot) \geq 0$ und somit $\gamma_\pi(\cdot) \geq 0$ auch in G_π.[7] □

Existiert ein $v'_0 \in V$ so, dass alle $v \in V$ von v'_0 aus erreichbar sind, so gilt die Behauptung auch direkt mit $\pi(\cdot) := \mathrm{d}(\cdot)$. In stark zusammenhängenden Graphen kann $v'_0 \in V$ beliebig gewählt werden.

Der so reduzierte Graph G_π kann nun zur SSSP-Suche mit beliebigem(!) Startknoten verwendet werden. Wir kommen darauf im Kapitel 7 über APSP-Algorithmen zurück.

Wir formulieren nun basierend auf Gleichung 3.9 und Lemma 3.10 einen sehr allgemeinen Kürzeste-Wege-Algorithmus. Zur Initialisierung des Algorithmus und Verwendung der Eigenschaft aus Gleichung 3.9 definieren wir zwei elementare Prozeduren und eine Funktion, die wir auch in anderen Algorithmen benutzen werden.

[7] Waren die Kantengewichte schon vorher alle nichtnegativ, so haben alle kürzesten Wege von v_0 in G' die Länge 0. Die dadurch gewählte Potentialfunktion lässt die Kantengewichte unverändert.

Prozedur 3.15 Init_SSSP(v_0)

$P_{3.15}.1$ $D(v_0) := 0$;
$P_{3.15}.2$ **for** $v \in V - \{v_0\}$ **do** $D(v) := \infty$ **od**

Funktion 3.16 is_decreasing(u, v)

$F_{3.16}.1$ **if** $D(v) > D(u) + \gamma(uv)$ **then return** true **else return** false **fi**

Prozedur 3.17 decrease_key(u, v)

$P_{3.17}.1$ $D(v) := D(u) + \gamma(uv)$

Algorithmus 3.18 *Generischer SSSP-Algorithmus*

$A_{3.18}.1$ Init_SSSP(v_0);
$A_{3.18}.2$ **while** $\exists$ uv: is_decreasing(u, v) **do** decrease_key(u, v) **od**

Solange es noch eine verkürzende Kante uv gibt (d. h. $D(v) > D(u) + \gamma(uv)$), setze man $D(v) := D(u) + \gamma(uv)$. Die Auswahl der Kante uv ist dabei nichtdeterministisch. Eine Kante uv ist genau dann verkürzend, wenn über uv der Knoten v auf einem kürzeren als dem bisher kürzesten betrachteten Weg erreicht werden kann.

Satz 3.19 (Korrektheit des generischen SSSP-Algorithmus) *Der Algorithmus 3.18 löst das SSSP-Problem für Graphen $G = (V, E, \gamma)$ ohne negative Zyklen für einen Startknoten $v_0 \in V$.*

Beweis: Wenn der Algorithmus terminiert, so folgt die Korrektheit direkt aus dem Lemma 3.10: Die $D(\cdot)$ nehmen nur Werte tatsächlicher Weglängen an ($D(\cdot) \geq d(\cdot)$); wird die while-Schleife $A_{3.18}.2$ verlassen, so sind alle Kanten uv konsistent und nach Lemma 3.10 gilt $D(\cdot) = d(\cdot)$. Es muss weiterhin $D(v_0) = 0$ gelten, sonst hätte es einen negativen Zyklus von v_0 aus gegeben und der Algorithmus wäre nicht terminiert. □

Hat der Graph negative Zyklen, so terminiert der Algorithmus nicht, da aufgrund des Lemmas 3.3 mit $\pi(\cdot) := D(\cdot)$ in jedem negativen Zyklus mindestens eine Kante uv Länge $\gamma_\pi(uv) < 0$ haben muss und diese somit nach Bemerkung 3.12 verkürzend ist. Es kann abgebrochen werden, wenn die Laufzeit die obere Schranke überschreitet oder wenn für einen Knoten v der Wert $D(v)$ kleiner als $-(n-1) \cdot \gamma_{|\max|}$ wird.

Bevor wir die Laufzeit abschätzen, zeigen wir, wie gleichzeitig auch der Kürzeste-Wege-Baum berechnet werden kann.

Lemma 3.20 (Kürzeste-Wege-Baum-Berechnung in $A_{3.18}$) *Merkt man sich im Schritt $A_{3.18}.2$ zu dem Knoten v den* Vorgänger $\mathrm{pred}(v) := u$ *(d. h., die Kante uv hat die Verringerung von $\mathrm{D}(v)$ bewirkt), so ergeben die Knoten v mit den (zuletzt) gemerkten Vorgängern $\mathrm{pred}(v)$ die Kanten $\mathrm{pred}(v)v$ eines Kürzeste-Wege-Baums. Der Aufwand ist proportional zur Anzahl der* decrease_key$(\cdot,\cdot)$*-Aufrufe.*[8]

Beweis: Angenommen, es gibt einen Weg $w = v_1 v_2 \cdots v_k$ mit $\mathrm{pred}(v_i) = v_{i-1}$, $i \in \{2,\ldots,k\}$ und die Kante $v_k v_1$ ist verkürzend, d. h., man könnte $\mathrm{D}(v_1) := \mathrm{D}(v_k) + \gamma(v_k v_1)$ und $\mathrm{pred}(v_1) := v_k$ setzen, sodass die Kanten $\mathrm{pred}(v_i)v_i$, $i \in \{1,\ldots,k\}$ einen Zyklus bilden. Die Länge dieses Zyklus $v_1 v_2 \cdots v_k v_1$ lässt sich somit wie folgt abschätzen: Es ist $\mathrm{pred}(v_i) = v_{i-1}$, d. h., $\mathrm{D}(v_i)$ hat über $\mathrm{D}(v_{i-1})$ und $\gamma(v_{i-1}v_i)$ seinen Wert erhalten und $\mathrm{D}(v_{i-1})$ kann sich danach noch verringert haben (jedoch nicht $\mathrm{D}(v_i)$). Es gilt $\mathrm{D}(v_{i-1}) + \gamma(v_{i-1}v_i) \leq \mathrm{D}(v_i)$. Weiterhin gilt $\mathrm{D}(v_k) + \gamma(v_k v_1) < \mathrm{D}(v_1)$, da die Kante $v_k v_1$ verkürzend ist. Somit gilt

$$\begin{aligned}
\mathrm{D}(v_1) + \gamma(v_1 v_2 \cdots v_k v_1) &= \mathrm{D}(v_1) + \gamma(v_1 v_2 \cdots v_k) + \gamma(v_k v_1) \\
&\leq \mathrm{D}(v_2) + \gamma(v_2 v_3 \cdots v_k) + \gamma(v_k v_1) \\
&\vdots \\
&\leq \mathrm{D}(v_k) + \gamma(v_k v_1) \\
&< \mathrm{D}(v_1)
\end{aligned}$$

womit $\gamma(v_1 v_2 \cdots v_k v_1) < 0$ und $v_1 v_2 \cdots v_k v_1$ ein negativer Zyklus ist.

Existiert kein negativer Zyklus, so sind nach Terminierung alle Kanten konsistent und für die Kanten $\mathrm{pred}(v_i)v_i$ gilt einerseits

$$\mathrm{D}(\mathrm{pred}(v_i)) + \gamma(\mathrm{pred}(v_i)v_i) \leq \mathrm{D}(v_i)$$

und andererseits nach Satz 3.19 $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$. Zusammen mit der Konsistenzeigenschaft $\mathrm{d}(v_i) \leq \mathrm{d}(\mathrm{pred}(v_i)) + \gamma(\mathrm{pred}(v_i)v_i)$ folgt

$$\mathrm{d}(v_i) = \mathrm{d}(\mathrm{pred}(v_i)) + \gamma(\mathrm{pred}(v_i)v_i)$$

und die $n-1$ Kanten $\mathrm{pred}(v_i)v_i$, $i \in \{1,\ldots,n-1\}$ sind zyklenfrei[9] und bilden nach Lemma 3.6 und Lemma 3.7 den Kürzeste-Wege-Baum. □

[8] Die Berechnung des Kürzeste-Wege-Baums mit Hilfe des Lemmas 3.7 ist unter Umständen effizienter, da nicht bei allen Algorithmen die Anzahl der decrease_key$(\cdot,\cdot)$-Aufrufe durch $O(m)$ beschränkt ist. Der asymptotische Gesamtaufwand für die Algorithmen erhöht sich jedoch nicht.

[9] Es ist weiterhin $\mathrm{pred}(v_0) = \mathrm{nil}$, da nach Annahme kein negativer Zyklus existiert.

Satz 3.21 (Laufzeit des generischen SSSP-Algorithmus) *Der Algorithmus 3.18 löst das SSSP-Problem für Graphen $G = (V, E, \gamma)$ ohne negative Zyklen für einen Startknoten $v_0 \in V$ in $O(m \cdot 2^{n \log n})$. Für ganzzahlige Kantengewichte $\gamma : E \to \mathbb{Z}$ ist mit $\gamma_{|\max|} := \max\{|\gamma(uv)| \mid uv \in E\}$ die Laufzeit zusätzlich durch $O(m \cdot n^2 \cdot \gamma_{|\max|})$ beschränkt.*

Beweis: Angenommen der Graph hat keine negativen Zyklen. Sei $\gamma_{|\max|}$ der Betrag des betragsmäßig größten Kantengewichts, so gilt für jeden doppelpunktfreien Weg w: $-(n-1) \cdot \gamma_{|\max|} \leq \gamma(w) \leq (n-1) \cdot \gamma_{|\max|}$. Unter der weiteren Annahme, dass die Kantengewichte ganzzahlig sind, folgt, dass jeder Knoten nur $O(n \cdot \gamma_{|\max|})$ verschiedene D(·)-Werte annehmen kann. In jedem Schleifendurchlauf $A_{3.18}.2$ wird für einen Knoten v der Wert D(v) um mindestens 1 verringert. Nach $O(n^2 \cdot \gamma_{|\max|})$ Schleifendurchläufen können keine verkürzenden Kanten mehr existieren, es sei denn, der Graph hat negative Zyklen.

Für den Fall, dass die Kantengewichte reellwertig sind, kann man über die durch die Vorgänger pred(·) definierten Wege-Bäume argumentieren. Die Werte D(·) sind durch die über die pred(·)-Zeiger definierten Wege gegeben. D. h., dass kein Wege-Baum mehrfach vorkommen kann, da sich die pred(·)-Zeiger nur durch Verringerung der D(·)-Werte verändern. Die Anzahl der Wege-Bäume ist demnach eine obere Schranke für die Anzahl der Iterationen. Für jeden Knoten kann der pred(·)-Zeiger n verschiedene Werte annehmen (kein Vorgänger oder einen der anderen $n-1$ Knoten). Der Startknoten hat (in Graphen ohne negative Zyklen) nie einen Vorgänger. Folglich gibt es höchstens $n^{n-1} \in O(2^{n \log n})$ verschiedene Wege-Bäume.

Das Finden einer verkürzenden Kante (falls so eine noch existiert) dauert $O(m)$, womit die Laufzeiten bewiesen sind. □

Für $m \in o(n \log n)$ gibt folgende Abschätzung eine leicht bessere Schranke: Jede maximal $(n-1)$-elementige Teilmenge der Kanten, deren Endpunkte paarweise verschieden sind, kann einen Wege-Baum bilden. Deren Anzahl ist höchstens $\sum_{i=0}^{n-1} \binom{m}{i} \in O(2^m)$ (man beachte, dass die Anzahl der Kanten in den Wege-Bäumen nicht von Beginn an schon $n-1$ beträgt).

Die Suche nach einer verkürzenden Kante ist mit $O(m)$ recht aufwendig. Eine konsistente Kante uv kann nur dann verkürzend werden, wenn sich durch ein decrease_key(u', u) der Wert D(u) verringert (zu Beginn sind nur die Kanten (v_0, v) verkürzend, da nur D(v_0) = 0 endlich ist). Statt einzeln nach verkürzenden Kanten zu suchen, ist es demnach natürlicher, sich die Knoten zu merken, von denen verkürzende Kanten ausgehen könnten, und statt einzelner Kanten alle von u ausgehenden Kanten zu prüfen, ob sie

verkürzend oder konsistent sind. Die Knoten u, die möglicherweise ausgehende verkürzende Kanten haben, werden in einer Menge R verwaltet (diese wird mit $R := \{v_0\}$ initialisiert). Dies führt zu dem *modifizierten generischen SSSP-Algorithmus*. Wir ersetzen $A_{3.18}.2$ durch

$A_{3.18}.2$ **while** $\exists\, u \in R$ **do**
$A_{3.18}.2.1$ wähle $u \in R$; $R := R - \{u\}$;
$A_{3.18}.2.2$ **for all** $uv \in E$ **do**
$A_{3.18}.2.2.1$ **if** is_decreasing(u, v) **then**
$A_{3.18}.2.2.1.1$ decrease_key(u, v); $R := R \cup \{v\}$ **fi od od**

Die Korrektheit folgt direkt aus dem Satz 3.19.

Satz 3.22 (Laufzeit des modifizierten gen. SSSP-Algorithmus) *Der modifizierte Algorithmus 3.18 löst das SSSP-Problem für Graphen $G = (V, E, \gamma)$ ohne negative Zyklen für einen Startknoten $v_0 \in V$ in $O(n \cdot 2^n)$. Für ganzzahlige Kantengewichte $\gamma : E \to \mathbb{Z}$ ist die Laufzeit zusätzlich durch $O(m \cdot n \cdot \gamma_{|\max|})$ beschränkt.*

Beweis: Wie im Beweis zu Satz 3.21 kann bei ganzzahligen Kantengewichten jeder Knoten u weniger als $2n \cdot \gamma_{|\max|}$ verschiedene $D(\cdot)$-Werte annehmen. Nur, wenn sich $D(u)$ verringert, wird u in R aufgenommen und im Folgenden die von u ausgehenden Kanten untersucht. Die Wahl eines Knotens u aus R geschieht nichtdeterministisch in $O(1)$. Die Gesamtlaufzeit beträgt somit

$$O(n \cdot \gamma_{|\max|} \cdot \sum_{u \in V} |\{uv \in E\}|) = O(n \cdot m \cdot \gamma_{|\max|})$$

Im Falle reellwertiger Kantengewichte zeigen wir eine obere Schranke von $O(2^n)$ Iterationen: O. B. d. A. können wir uns hier auf nichtnegative Kantengewichte beschränken, da nach Lemma 3.14 stets eine Potentialfunktion $\pi(\cdot)$ so existiert, dass die reduzierten Kantengewichte alle nichtnegativ sind. Aufgrund der Invarianz bzgl. der Ordnung der Weglängen (Lemma 3.5) werden dieselben Kanten in derselben Reihenfolge verkürzend sein, sodass der Ablauf des Algorithmus in G_π identisch dem eines Worst-Case-Beispiels in G sein kann.

Wir zeigen nun: Wird der Wert $D(v_k)$ aufgrund des Weges $w = v_0 v_1 \cdots v_k$ auf den Wert $D(v_k) := \gamma(w)$ verringert, so kann keine Permutation τ (mit $\tau(0) = 0$) ungleich der Identität existieren, sodass im weiteren Verlauf auch der Weg $w' = v_0 v_{\tau(1)} \cdots v_{\tau(k)}$ den Wert $D(v_{\tau(v_k)})$ auf $\gamma(w')$ verändert (v_k und $v_{\tau(k)}$ können verschieden sein). Sei μ der größte Index, sodass $\tau(i) = i$, $i \in \{0, \ldots, \mu\}$ ($\mu < k$, da $\tau(\cdot)$ nicht die Identität ist). Damit sind $v_\mu v_{\mu+1}$ und

$v_\mu v_{\tau(\mu+1)}$ zwei verschiedene Kanten in G. Der Wert $\mathrm{D}(v_\mu)$ sei auf $\gamma(v_0 \cdots v_\mu)$ gesetzt. Wird v_μ bearbeitet, so werden

$$\mathrm{D}(v_{\mu+1}) := \mathrm{D}(v_\mu) + \gamma(v_\mu v_{\mu+1})$$

und

$$\mathrm{D}(v_{\tau(\mu+1)}) := \mathrm{D}(v_\mu) + \gamma(v_\mu v_{\tau(\mu+1)})$$

gesetzt, da sonst w und w' nicht mehr zum Setzen von $\mathrm{D}(v_k)$ bzw. $\mathrm{D}(v_{\tau(k)})$ benutzt werden könnten. Sei $j := \mu + 1$, $j' := \tau(\mu + 1)$ ($j' > \mu + 1$, da $\tau(i) = i,\ i \leq \mu$ und $\tau(\mu + 1) \neq \mu + 1$), $w_{jj'} := v_j v_{j+1} \cdots v_{j'}$, so muss

$$\gamma(v_\mu v_{\mu+1}) + \gamma(w_{jj'}) < \gamma(v_\mu v_{\tau(\mu+1)})$$

sein, da sonst die Kante $v_{j'-1}v_{j'}$ nicht verkürzend wäre und so der Weg w später nicht den Wert $\mathrm{D}(v_k) := \gamma(w)$ setzen kann. Sei j'' der Wert mit $\tau(j'') = \mu + 1 =: j$ ($j'' > \mu + 1$, da $\tau(i) = i,\ i \leq \mu$ und $\tau(\mu + 1) \neq \mu + 1$), $w_{j'j} := v_{\tau(\mu+1)} v_{\tau(\mu+2)} \cdots v_{\tau(j'')}$, so muss

$$\gamma(v_\mu v_{\tau(\mu+1)}) + \gamma(w_{j'j}) < \gamma(v_\mu v_{\mu+1})$$

sein, da sonst die Kante $v_{\tau(j''-1)}v_{\tau(j'')}$ nicht verkürzend wäre und so der Weg w' später nicht den Wert $\mathrm{D}(v_{\tau(k)}) := \gamma(w')$ setzen kann. Durch Addition dieser beiden Ungleichungen folgt, dass

$$\gamma(w_{jj'}) + \gamma(w_{j'j}) < 0$$

und somit diese beiden Wege zusammen einen negativen Zyklus bilden, ein Widerspruch dazu, dass G nur nichtnegative Kantengewichte hat.

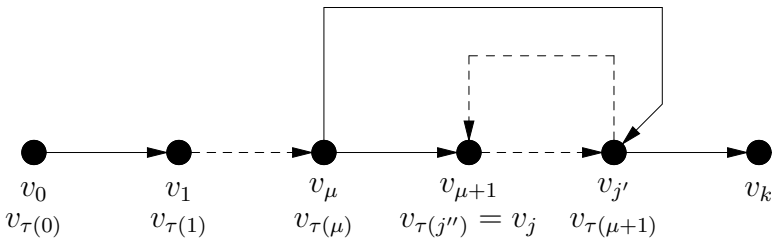


Abbildung 3.2: Illustration des Beweises zu Satz 3.22

Jede Teilmenge der Knoten $\{v_1, \ldots, v_k\}$ kann also nur in einer Permutation $\tau(\cdot)$ zusammen mit dem Startknoten v_0 einen Weg $w = v_0 v_{\tau(1)} v_{\tau(2)} \cdots v_{\tau(k)}$ bilden, der zur Verringerung der Entfernung $\mathrm{D}(v_{\tau(k)})$ beiträgt.

Ferner gilt: Setzt $w = v_0 v_1 \cdots v_k$ den Wert $\mathrm{D}(v_k) := \gamma(w)$, so sind dadurch auch die Permutationen und damit die Wege für die Teilmengen $\{v_0, \ldots, v_i\}$

auf $w_i = v_0 v_1 \cdots v_i$ festgelegt. In jeder Iteration wird so eine der Teilmengen benutzt und kann somit später nicht nochmals verwendet werden.

Die Anzahl der Teilmengen (bei der v_0 fest dabei ist) beträgt 2^{n-1} und bestimmt eine obere Schranke für die Anzahl der Iterationen. In jeder dieser Iterationen werden höchstens n ausgehende Kanten bearbeitet, womit die Gesamtlaufzeit bewiesen ist. □

Wir werden im nächsten Abschnitt auch Beispiele vorstellen, die tatsächlich 2^{n-1} Iterationen benötigen.

[AMO93] behaupten die Schranke $O(2^n)$ Iterationen auch für den (nicht modifizierten) generischen SSSP-Algorithmus, beziehen sich dabei aber auf [GP86], die diese Schranke (ohne Beweis) nur für die modifizierte Variante angeben. [Tar83] gibt die $O(2^m)$-Schranke an – aber ebenfalls ohne Beweis. In allen weiteren uns vorliegenden Arbeiten wird für eine $O(2^n)$ Schranke stets direkt oder auf Umwegen auf [GP86] verwiesen, sodass die exakte Komplexität des generischen Algorithmus nach unserem Kenntnisstand noch offen ist – die Anzahl der Iterationen kann zwar 2^{n-1} übersteigen, bleibt aber in allen uns bekannten und von uns konstruierten Beispielen durch $O(2^n)$ beschränkt.

Im generischen Algorithmus können dieselben Knoten auch in mehreren Permutationen auftreten. In dem Graphen $G = (\{v_0, v_1, v_2\}, \{v_0v_1, v_0v_2, v_1v_2, v_2v_1\}, \gamma)$ sind zunächst für den Weg $w = v_0v_1v_2$ die Kanten mit $\gamma(v_0v_1) = 1$ und $\gamma(v_1v_2) = 1$ verkürzend und danach für den Weg $w' = v_0v_2v_1$ die Kanten mit $\gamma(v_0v_2) = 0$ und $\gamma(v_2v_1) = 0$, ohne dass G einen negativen Zyklus hat.[10]

3.3 Entfernungskorrigierende Instanzen

Wir betrachten nun drei konkrete Ausprägungen, wie die Menge R verwaltet wird und welcher Knoten u als nächstes in $A_{3.18}$.2.1/2.2 betrachtet wird. Dies führt auf die Algorithmen von Bellmann-Ford ([Bel58], [For56], Verwaltung von R als Warteschlange), D'Esopo-Pape ([PW60], [Pap74], [Pap80], [Pap83], Verwaltung von R als zweiseitige Warteschlange) und Pallottino ([Pal84], Verwaltung von R als zwei getrennte Warteschlangen).

[10] Fügt man noch Kanten v_1v_0' und v_2v_0' hinzu, so kann man dies als Block mehrfach hintereinander hängen. Haben im i-ten Block die Kanten die Längen $\gamma(v_{0,i}v_{1,i}) = \gamma(v_{1,i}v_{2,i}) = \gamma(v_{2,i}v_{0,i+1}) = 2^{-i}$, sonst $\gamma(\cdot) = 0$, dann kann die Knotenmenge V in $\Theta(2^{n/3})$ Permutationen den Wert $D(\cdot)$ des letzten Knotens der Kette setzen. Obwohl hier schon eine einzelne Teilmenge der Knoten exponentiell oft auftritt, bleibt der Gesamtaufwand auch in diesem Beispiel durch $O(2^n)$ Iterationen beschränkt.

3.3.1 Der Algorithmus von Bellmann-Ford und andere FIFO-Varianten

Der Algorithmus von Bellmann-Ford ([Bel58], [For56]) verwaltet die Menge R als Warteschlange (*First-in First-out* / FIFO).[11] Dadurch läuft der Algorithmus in *Phasen* ab. In der ersten Phase werden die ausgehenden Kanten des Startknotens betrachtet, in jeder weiteren Phase i jeweils die ausgehenden Kanten derjenigen Knoten v, deren Wert $\mathrm{D}(v)$ sich in der Phase $i-1$ verringert hat. Innerhalb einer Phase bräuchte das starre FIFO-Konzept nicht durchgehalten zu werden, um die Korrektheit und Laufzeit zu erhalten. [GKP85] haben dies mit dem „partitioning shortest path algorithm“ ausgeführt.

Offensichtlich sind im generischen SSSP-Algorithmus die Werte $\mathrm{D}(v)$ für alle Knoten v streng monoton fallend. Das heißt insbesondere: Wird für einen Knoten v der Wert $\mathrm{D}(v) := \mathrm{d}(v)$ gesetzt, so wird v zum letzten Mal in die Menge R eingefügt und alle in v eingehenden Kanten bleiben fortan konsistent. Wir wollen diese Eigenschaft für den Bellman-Ford-Algorithmus nun genauer fassen:

Lemma 3.23 (Phasen-Eigenschaft des Bellman-Ford-Algorithmus) *Sei $w = v_0 v_1 \cdots v_k$ ein beliebiger Weg von v_0 nach v_k mit k Kanten, so gilt nach der k-ten Phase $\mathrm{D}(v_k) \leq \gamma(w)$.*

Beweis: Die Aussage gilt für $k = 0$ (der Weg v_0 (ohne Kanten) hat Länge $\gamma(v_0) = 0$). Mit vollständiger Induktion schließen wir nun: Nach der $(k-1)$-ten Phase gelte $\mathrm{D}(v_{k-1}) \leq \gamma(v_0 \cdots v_{k-1})$. Der Wert $\mathrm{D}(v_{k-1})$ wurde in der j-ten Phase zuletzt verringert, $j \leq k-1$, und v_{k-1} in R aufgenommen. Somit wird die Kante $v_{k-1}v_k$ in der $(j+1)$-ten Phase betrachtet, $j+1 \leq k$ (oder v_{k-1} war schon in R und $v_{k-1}v_k$ wird ggf. noch in Phase j bearbeitet). Entweder gilt $\mathrm{D}(v_k) \leq \gamma(w)$ schon vorher, anderenfalls ist mit

$$\mathrm{D}(v_k) > \gamma(w) = \gamma(v_0 \cdots v_{k-1}) + \gamma(v_{k-1}v_k) \geq \mathrm{D}(v_{k-1}) + \gamma(v_{k-1}v_k)$$

$v_{k-1}v_k$ verkürzend und danach $\mathrm{D}(v_k) = \mathrm{D}(v_{k-1}) + \gamma(v_{k-1}v_k) \leq \gamma(w)$. □

[11] Befindet sich ein Knoten v schon in der Warteschlange, so wird er weder erneut eingefügt (dies wäre unnötig) noch die Position in der Warteschlange verändert (dies würde für die FIFO statt einer einfach-verketteten Liste eine doppelt-verkettete Liste erfordern, um das Element an der aktuellen Stelle zu entfernen). [SW81] untersuchen u.a. den Einfluss des (Nicht-)Verschiebens auf das Verhalten des Algorithmus, wobei nichts für eine Veränderung der Position in der FIFO spricht.

Ist $w = v_0 \cdots v_k$ kürzester Weg, so folgt mit $\mathrm{D}(v_k) \geq \mathrm{d}(v_k)$ und Lemma 3.23, dass nach k Phasen $\mathrm{D}(v_k) = \mathrm{d}(v_k)$ gilt.

Sei $t := \max_{v \in V}(\min_{w=v_0 \cdots v:\ \gamma(w)=\mathrm{d}(v)}(|w|))$ die Mindestanzahl der Kanten, sodass jeder Knoten v auf einem kürzesten Weg von v_0 aus mit t Kanten erreichbar ist, so sind nach t Phasen im Bellman-Ford-Algorithmus die $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ berechnet.[12]

Satz 3.24 (Korrektheit & Laufzeit des Bellman-Ford-Algorithmus) *Der Bellman-Ford-Algorithmus berechnet in $O(t \cdot m)$ Schritten die kürzesten Wege oder erkennt einen negativen Zyklus in $O(n \cdot m)$ Schritten. Ist t vorab bekannt, so wird ein negativer Zyklus in $O(t \cdot m)$ Schritten erkannt.*

Beweis: Die Korrektheit und Laufzeit folgt direkt mit dem Satz 3.19 und Lemma 3.23. In jeder Phase wird jede Kante nur einmal betrachtet, jede Phase benötigt somit $O(m)$ Schritte. Zu jedem Knoten existiert ein kürzester Weg von v_0 mit höchstens $t \leq n-1$ Kanten und nach $t \leq n-1$ Phasen gilt in Graphen ohne negative Zyklen $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$. Wird in Phase $t+1 \leq n$ eine verkürzende Kante uv entdeckt, so existiert ein Weg $w = v_0 \cdots v$ mit $t+1 \leq n$ Kanten, der kürzer ist als jeder Weg $w' = v_0 \cdots v$ mit höchstens $t \leq n-1$ Kanten. Somit muss ein negativer Zyklus vorliegen (vgl. Lemma 3.23 und Fußnote 6, Seite 32). □

Der Algorithmus kann auch nach weniger als t Phasen korrekt terminieren (siehe das Beispiel in Abbildung 3.3 mit $t = 5$). Das liegt daran, dass der Knoten v z. B. durch Betrachten der Kante uv in Phase $i-1$ in die Warteschlange kam und vor Bearbeiten des Knotens v in Phase i der Wert $\mathrm{D}(v)$ nochmals verringert wurde. Im Beispiel in Abbildung 3.3 wird in Phase 1 nur der Startknoten v_0 betrachtet und die Knoten v_1 bis v_5 in dieser Reihenfolge in die Warteschlange eingefügt. Wird nun der Knoten v_1 in Phase 2 betrachtet, so wird $\mathrm{D}(v_2) := 0$ gesetzt (v_2 befindet sich bereits in der Warteschlange, wird also nicht hinten eingefügt). Im Verlauf der Phase 2 werden nun alle $\mathrm{D}(v_i) := 0$ gesetzt und der Algorithmus terminiert (unabhängig von der Anzahl der Knoten) am Ende von Phase 2.

In [MN99] wird ein Konstruktionsverfahren angegeben, das für beliebige n und m einen Graphen erzeugt, sodass der Bellmann-Ford Algorithmus Laufzeit $\Theta(nm)$ hat.[13] In der Praxis wird dieser Worst-Case aber fast nie erreicht ([CGR96]).

[12] Das t entspricht der Tiefe des Kürzeste-Wege-Baums, wenn dieser nach Lemma 3.7 mittels Breitensuche bestimmt wird.

[13] Die Graphen sind dabei sehr unregelmäßig, ein Teil der Knoten hat nur 1 oder 2 ausgehende Kanten, die restlichen Knoten sind paarweise verbunden.

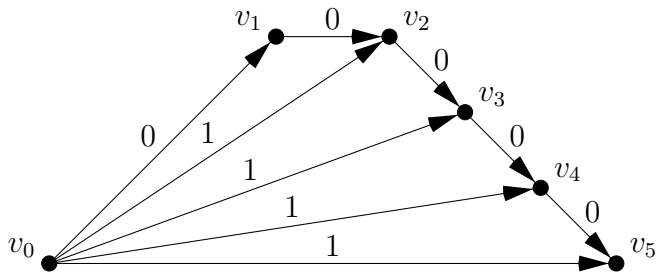


Abbildung 3.3: Beispiel für Bellman-Ford mit weniger als t Phasen

Bemerkung 3.25 (Bellman-Ford mit Vorgängerheuristik) *Wird in $A_{3.18}$.2.1 der Knoten $u \in R$ gewählt und befindet sich der Vorgänger $u' = \mathrm{pred}(u)$ ebenfalls in R, so wurde $\mathrm{D}(u')$ verringert, nachdem u (zuletzt) in R eingefügt wurde.*[14] *Demnach ist es nicht sinnvoll, die ausgehenden Kanten von u zu untersuchen, da sich $\mathrm{D}(u)$ z. B. durch die Kante $u'u$ nochmals verringern wird und die Kanten uv (und ggf. die Nachfolger von v) nochmals betrachtet werden müssten ([CGR96]).*[15] *Die Vorgängerheuristik benötigt also nur eine zusätzliche* **if***-Abfrage in einem geeignet verwalteten Boolean-Array, um festzustellen, ob $\mathrm{pred}(u) \in R$ gilt. Das asymptotische Worst-Case-Verhalten verändert sich dadurch nicht. Im Mittel werden unnötige Schritte vermieden, [CGR96] haben z. B. für gitterähnliche Graphen eine Ersparnis von knapp 50% experimentell ermittelt. Für Graphen, bei denen die Vorgänger der Knoten sich selten oder nie gleichzeitig in der Warteschlange befinden, erhöht sich die Laufzeit minimal ($\ll 5\%$).*

[MN99] beschreibt eine weiterführende Idee von Tarjan: Wird der Wert $\mathrm{D}(u)$ verringert, so ist nicht nur die Abarbeitung eines direkten Nachfolgers v überflüssig, sondern die Knoten des gesamten Unterbaums brauchen nicht weiter betrachtet zu werden, da sich die Werte aller dieser Knoten in den folgenden Phasen verringern würden. Verringert die Kante uv den Wert $\mathrm{D}(v)$ und v ist ein Vorfahre von u im bisher berechneten Kürzeste-Wege-Baum, so ist damit ein negativer Zyklus entdeckt, da die reduzierten Gewichte der Kanten des Kürzeste-Wege-Baums 0 sind und die verkürzende Kante uv negative reduzierte Länge hat (vgl. auch Beweis zu Lemma 3.20). Dies ermöglicht negative Zyklen frühzeitig zu erkennen, während der normale Bellman-Ford-Algorithmus negative Zyklen nur dadurch erkennt, dass die Anzahl der Pha-

[14] Ob dies in der gleichen Phase geschehen ist, in der $\mathrm{D}(u)$ verringert wurde, oder in der aktuellen Phase, kann man nicht ohne Weiteres feststellen. Dieses beobachtet man auch im Beispiel in Abbildung 3.3.

[15] Ein direktes erneutes Einfügen von u ist nicht nötig.

sen $n-1$ übersteigt. Um auch die Knoten aus den Unterbäumen schnell ermitteln zu können, verwaltet diese Variante neben den Vorgängern den Kürzeste-Wege-Baum explizit (in Preorder) – trotz dieses Overheads ist die Laufzeit weiterhin durch $O(nm)$ beschränkt, in der Praxis ist die Variante von Tarjan meist schneller und nie wesentlich langsamer als die einfache Bellman-Ford-Variante. Ein Vergleich mit der Vorgängerheuristik ist uns nicht bekannt.

3.3.2 Der Algorithmus von D'Esopo und Pape

Eine vermeintliche Schwäche des Bellman-Ford-Algorithmus ist die starre Breitensuche. Dadurch können Korrekturen der $\mathrm{D}(\cdot)$-Werte nur durch komplettes Wiederholen der Schritte aus vorhergehenden Phasen im Graphen verbreitet werden. Wird für den Knoten v der Wert $\mathrm{D}(v)$ verringert, so scheint es sinnvoll, zunächst die Werte $\mathrm{D}(\cdot)$ der von dort schon einmal betrachteten Knoten zu korrigieren, bevor man die Kürzeste-Wege-Suche bei Knoten fortsetzt, die noch gar nicht betrachtet wurden. Diese Idee wurde erstmals von D'Esopo (in [PW60]) formuliert und von Pape ([Pap74], [Pap80], [Pap83]) im Weiteren untersucht. Die Menge der Knoten R, deren ausgehende Kanten im Algorithmus 3.18 noch untersucht werden müssen, werden jetzt mit einer Warteschlange verwaltet, in die die Knoten sowohl am Ende als auch am Anfang eingefügt werden können. Dieses kann man als Kopplung eines Kellers (Last-in First-out / LIFO) und einer Warteschlange (FIFO) auffassen.

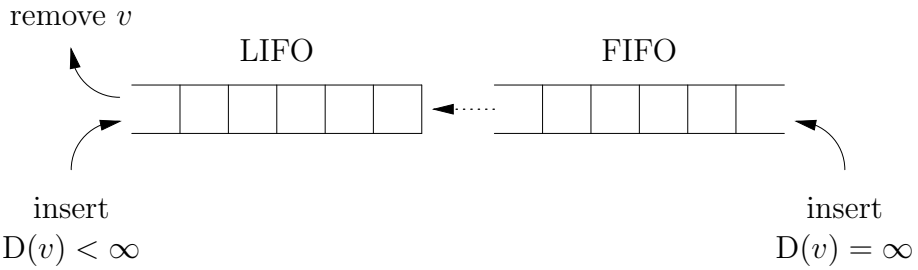


Abbildung 3.4: Die Datenstruktur im D'Esopo-Pape-Algorithmus

Knoten, die erstmals eingefügt werden (das sind diejenigen v mit $\mathrm{D}(v) = \infty$), werden in die (hintere) Warteschlange eingefügt, Knoten, deren ausgehende Kanten schon einmal betrachtet wurden ($\mathrm{D}(v)$ hat bereits einen endlichen Wert), werden vorne in den Keller eingefügt. So werden die Werte $\mathrm{D}(\cdot)$ der zu v adjazenten Knoten korrigiert, bevor neue Knoten erstmals bearbeitet werden. Im Algorithmus wird in Schritt $\mathrm{A}_{3.18}$.2.1 der Knoten aus dem Keller

entnommen, falls dieser nicht leer ist, ansonsten aus der (hinteren) Warteschlange.[16]

In der Anwendung zeigt sich dieser Algorithmus in vielen realen Fällen den meisten anderen Algorithmen überlegen ([CGR96], [ZN98], [ZN00]). In Fällen, in denen die eingeschränkte Dreiecksungleichung $\gamma(uw) \leq \gamma(uv) + \text{dist}(v, w)$ häufig verletzt wird, tendiert der D'Esopo-Pape-Algorithmus jedoch dazu, sehr langsam zu laufen. Dann führen oft Kanten mit hohem Gewicht, die später nicht zum Kürzeste-Wege-Baum gehören, in noch nicht bearbeitete Teile des Graphen und verursachen so häufige Korrekturen. Gilt $\gamma(uw) > \gamma(uv) + \text{dist}(v, w)$ und sind beide Kanten uv und uw verkürzend, so werden die Schritte von w aus umsonst gemacht, da später $\text{D}(w)$ über den Weg über v wieder verringert wird.

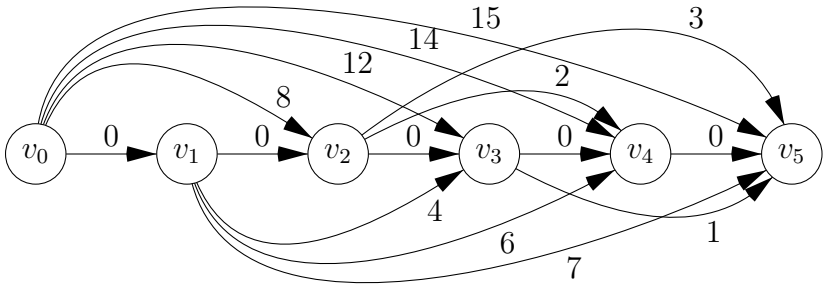


Abbildung 3.5: Worst-Case-Beispiel 1 für den D'Esopo-Pape-Algorithmus

[Ker81] und [AMO93] geben das Worst-Case-Beispiel in Abbildung 3.5 an, das diese Eigenschaft demonstriert und den Worst-Case von 2^{n-1} Iterationen des (modifizierten) generischen Algorithmus (Satz 3.22) erreicht. Selbst für dünne Graphen, d. h. $m \in O(n)$, geben [SW81] Beispiele an, die $O(2^{\frac{n}{2}})$ Iterationen durchführen.

Lemma 3.26 (Laufzeit des D'Esopo-Pape-Algorithmus) *Sei $G = (V, E, \gamma)$ mit $V = \{v_0, v_1, \ldots, v_n\}$, $E = \{v_i v_j \mid i < j\}$ und $\gamma(v_i v_j) = \sum_{k=i+2}^{j} 2^{n-k}$. Der D'Esopo-Pape-Algorithmus benötigt zur Lösung des SSSP-Problems für den Graphen in Abbildung 3.5 $\Theta(2^n)$ Schleifendurchläufe.*[17]

[16] Die Trennung der zweiseitigen Warteschlange in LIFO und FIFO ist in der Implementierung nicht nötig und wird hier nur zur Verdeutlichung der Ähnlichkeit mit dem im Folgeabschnitt vorgestellten Algorithmus nach Pallottino vorgenommen. Die Fallunterscheidung bei der Auswahl entspricht stets der Wahl des ersten Elements der zweiseitigen Warteschlange.

[17] Damit dieser Worst-Case erreicht wird, müssen in der Adjazenzliste die vom Start-

Beweis: Wir zeigen zunächst mit Hilfe der vollständigen Induktion über i, dass der Zustand $\mathrm{D}(v_j) = c + \sum_{k=i+1}^{j} 2^{n-k},\ j \in \{i, \ldots, n\}$ mit Kellerinhalt $v_n, v_{n-1}, \ldots, v_i$ nach $2^{n+1-i} - 1$ Schleifendurchläufen in dem Zustand $\mathrm{D}(v_j) = c,\ j \in \{i, \ldots, n\}$ mit leerem Keller endet:

Für $i = n$ gilt die Aussage. $\mathrm{D}(v_n) = c$ mit Kellerinhalt v_n geht nach einem Schleifendurchlauf in $\mathrm{D}(v_n) = c$ mit leerem Keller über, da v_n keine (verkürzenden) ausgehenden Kanten hat. Wir schließen nun von i auf $i - 1$: Zu Beginn gelte mit einer beliebigen Konstante c

$$\mathrm{D}(v_j) = c + \sum_{k=i}^{j} 2^{n-k},\ j \in \{i-1, \ldots, n\}$$

mit Kellerinhalt $v_n, v_{n-1}, \ldots, v_i, v_{i-1}$. Leicht umformuliert lautet dies

$$\mathrm{D}(v_{i-1}) = c\ \wedge\ \mathrm{D}(v_j) = c + 2^{n-i} + \sum_{k=i+1}^{j} 2^{n-k},\ j \in \{i, \ldots, n\}$$

Auf den Kellerinhalt $v_n, v_{n-1}, \ldots, v_i$ lässt sich nun die Induktionsvoraussetzung anwenden. Wir erhalten den Zustand

$$\mathrm{D}(v_{i-1}) = c\ \wedge\ \mathrm{D}(v_j) = c + 2^{n-i},\ j \in \{i, \ldots, n\}$$

Als Kellerinhalt verbleibt v_{i-1}. Nun sind alle Kanten $v_{i-1}v_j$ verkürzend, da $\sum_{k=i+1}^{n} 2^{n-k} < 2^{n-i}$. Wir erhalten

$$\mathrm{D}(v_{i-1}) = c\ \wedge\ \mathrm{D}(v_j) = c + \sum_{k=i+1}^{j} 2^{n-k},\ j \in \{i, \ldots, n\}$$

mit dem Kellerinhalt $v_n, v_{n-1}, \ldots, v_i$, sodass durch erneute Anwendung der Induktionsvoraussetzung

$$\mathrm{D}(v_j) = c,\ j \in \{i-1, \ldots, n\}$$

mit dann leerem Keller folgt. Die Anzahl der Schleifendurchläufe ist $S(v_n) = 1$ und $S(v_n, \ldots, v_{i-1}) = 2 \cdot S(v_n, \ldots, v_i) + 1$, somit $S(v_n, \ldots, v_1) = 2^n - 1$.

knoten v_0 ausgehenden Kanten in der Reihenfolge $v_n,\ v_{n-1},\ \ldots,\ v_1$ genannt sein, bei den anderen Knoten v_i folgen die adjazenten Knoten bzgl. dem Index in aufsteigender Reihenfolge $v_{i+1},\ v_{i+2},\ \ldots,\ v_n$.

Nach der Initialisierung wird der Knoten v_0 betrachtet, die Kanten v_0v_j sind alle verkürzend, es wird $\mathrm{D}(v_j) := \sum_{k=2}^{j} 2^{n-k}$ gesetzt und $v_n, \ldots, v_1$ in die Warteschlange eingefügt. Da nun jeder Knoten einen endlichen $\mathrm{D}(\cdot)$-Wert hat, verhält sich die Kopplung aus Keller und Warteschlange fortan wie ein Keller. Die Induktionsvoraussetzung ist erfüllt, die Gesamtanzahl der Schleifendurchläufe beträgt somit 2^n. □

Der Worst-Case aus Satz 3.22 mit $2^{|V|-1}$ Schleifendurchläufen – $|V|$ ist hier ausnahmsweise $n + 1$ – wird exakt erreicht. Jede Teilmenge der Knoten $\{v_1, \ldots, v_n\}$ wird zusammen mit dem Startknoten v_0 einmal zum Setzen eines $\mathrm{D}(\cdot)$-Wertes benutzt. Fügt man noch Kanten von v_n zu allen anderen Knoten mit Gewicht $\gamma(v_nv_i) = 1$ hinzu, so wird in den 2^{n-1} Iterationen, in denen die von v_n ausgehenden Kanten betrachtet werden, ein Aufwand von je $O(n)$ benötigt. Diese Kanten sind nie verkürzend, führen also zu keiner Veränderung des Ablaufs. Die Gesamtlaufzeit beträgt somit $O(n \cdot 2^n)$.

[Ker81] schlägt vor, die Kanten in der Adjazenzliste so anzuordnen, dass der Worst-Case vermieden wird. Im Beispiel in Abbildung 3.5 wird stets die Kante zuerst betrachtet, die das größte Kantengewicht hat. Würde man die Kanten in der Adjazenzliste andersherum anordnen (also jeweils die Kante mit kleinstem Gewicht zuerst betrachten), so würden die ausgehenden Kanten von jedem Knoten nur einmal betrachtet![18]

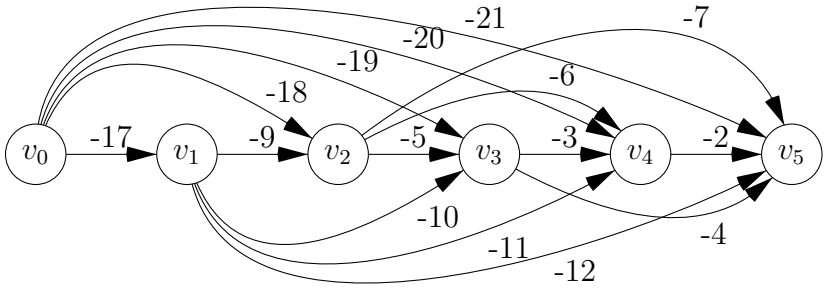


Abbildung 3.6: Worst-Case-Beispiel 2 für den D'Esopo-Pape-Algorithmus

Betrachtet man im Beispiel in Abbildung 3.6 ebenfalls die Kanten mit kleinstem Gewicht zuerst, so führt dies zum Worst-Case, während die absteigende Reihenfolge der Gewichte nur lineare Zeit benötigt. Bei beiden Beispielen führt die Reihenfolge mit den betragsmäßig kleinsten Gewichten zuerst auf eine optimale Reihenfolge. Aber auch für diese lässt sich ein Worst-Case-Beispiel konstruieren (Abbildung 3.7).

[18] Es reicht hier, die Kanten des Startknoten andersherum anzuordnen, die Knoten werden dann in der Reihenfolge der ansteigenden Indizes betrachtet. Die verkürzenden Kanten betreffen jeweils nur Knoten, die sich noch in der Warteschlange befinden.

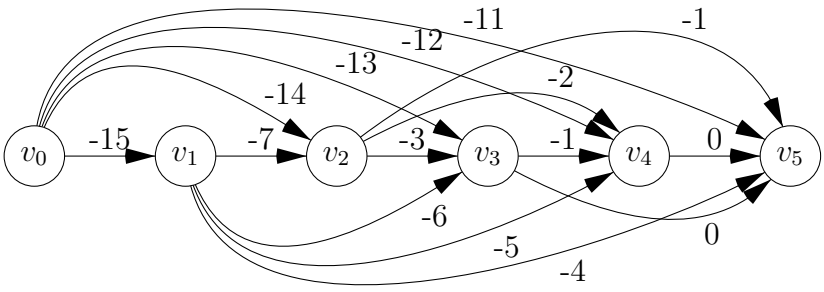


Abbildung 3.7: Worst-Case-Beispiel 3 für den D'Esopo-Pape-Algorithmus

Alle diese Beispiele lassen sich durch verschiedene Potentialfunktionen aus dem Beispiel aus Abbildung 3.8 herleiten, bei dem jeder Versuch, die ausgehenden Kanten nach Gewicht anzuordnen, vergebens ist. Sei der Graph $G = (V, E, \gamma)$ mit den Knoten $V = \{v_0, \ldots, v_n\}$, den Kanten $E = \{v_i v_j \mid i < j\}$ und Gewichtsfunktion $\gamma(v_i v_j) = -2^{n-1-i}$. Im Folgenden sind die Potentialfunktionen $\pi(\cdot)$ angegeben, mit denen sich aus Beispiel 4 die anderen Beispiele ableiten lassen:

Beispiel 1: Man wähle $\pi(v_i) := 2^n - 2^{n-i}$.[19]

Beispiel 2: Man wähle $\pi(v_i) := i$.

Beispiel 3: Man wähle $\pi(v_i) := -i$.

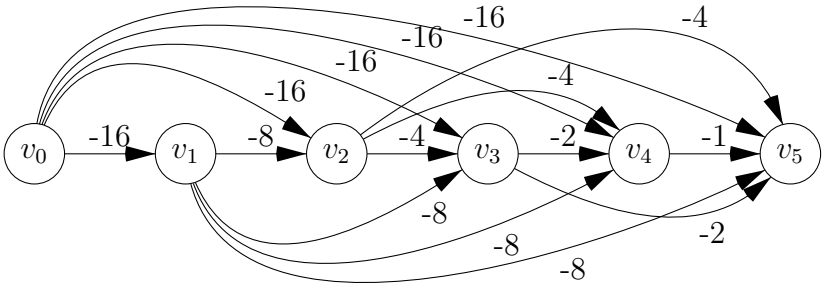


Abbildung 3.8: Worst-Case-Beispiel 4 für den D'Esopo-Pape-Algorithmus

An diesen Beispielen lässt sich auch gut demonstrieren, dass sich – wie im Lemma 3.5 beschrieben – die Ordnung der Weglängen durch Anwendung von

[19]Da die reduzierten Kantengewichte unabhängig von additiven Konstanten in $\pi(\cdot)$ sind, reicht auch $\pi(v_i) := -2^{n-i}$, wobei dann aber $\pi(v_0)$ nicht mehr null ist - dies ist aber nur eine Frage der Ästhetik.

Potentialfunktionen nicht ändert. Die Reihenfolge der Schritte bleibt gleich. Kanten sind unabhängig von der verwendeten Potentialfunktion immer zum selben Zeitpunkt im Ablauf der Algorithmen verkürzend.[20]

Eine interessante Eigenschaft des D'Esopo-Pape-Algorithmus gegenüber den FIFO-Varianten aus Abschnitt 3.3.1 ist, dass zu dem Zeitpunkt, an dem die vom Knoten v ausgehenden Kanten betrachtet werden, der Wert $\mathrm{D}(v)$ der Weglänge des Weges entspricht, der durch die gespeicherten Vorgänger zur Wurzel eindeutig definiert ist. Bei den FIFO-Varianten kann sich der $\mathrm{D}(\cdot)$-Wert eines Knotens auf diesem Weg und damit die Weglänge zum Knoten v verringert haben, sodass in letzterem Fall die von v ausgehenden Kanten später in jedem Fall nochmals betrachtet werden müssen.

Lemma 3.27 *Im D'Esopo-Pape-Algorithmus gilt in Graphen ohne negative Zyklen für den Knoten v, der zuletzt aus der zweiseitigen Warteschlange entnommen wurde, dass die Länge des Weges $w = \mathrm{pred}^i(v) \cdots v,\ \mathrm{pred}^i(v) = v_0$ gleich dem Wert $\mathrm{D}(v)$ ist.*

Beweis: Nach Lemma 3.20 können die $\mathrm{pred}(\cdot)$-Zeiger in Graphen ohne negative Zyklen keine Zyklen bilden, sodass für jeden Knoten ein eindeutiges i existiert, sodass $\mathrm{pred}^i(v) = v_0$ ($\mathrm{pred}(v_0) = \mathrm{nil}$).

Für v_0 gilt die Eigenschaft des Lemmas. Wir folgern durch Induktion: Hat sich seit dem Einfügen des Knotens v (mit dem Wert $\mathrm{D}(v)$) der Wert $\mathrm{D}(u)$ für einen Knoten u, der auf dem zu dem Wert $\mathrm{D}(v)$ zugehörigen Weg liegt, – und damit auch die Länge des Weges zu v – verringert, so wurde u auf den Keller gelegt, da $\mathrm{D}(u)$ bereits einen endlichen Wert hatte – sonst kann u nicht auf dem Weg nach v gelegen haben. Somit wird u vor v betrachtet und so ebenso die Knoten auf dem Weg von u nach v, sodass $\mathrm{D}(v)$ wieder den dem Weg entsprechend korrigierten Wert annimmt, bevor die von v ausgehenden Kanten betrachtet werden. □

Die Kopplung von Keller und Warteschlange bei D'Esopo-Pape lässt auch folgende Interpretation zu: Der Algorithmus arbeitet mit den auf dem Keller liegenden Knoten solange, bis von der Warteschlange ein weiterer Knoten hinzugenommen wird. Zu diesem Zeitpunkt ist jeweils ein Kürzeste-Wege-Baum für die bisher betrachteten Knoten berechnet mittels eines Algorithmus, der nur einen Keller benutzt. Die von dem neu hinzugenommenen Knoten ausgehenden Kanten verändern ggf. den bisher berechneten Kürzeste-Wege-Baum, dessen Knoten in den Keller aufgenommen und dann bearbeitet werden, bis

[20] Dies gilt nicht mehr, wenn die Auswahl der Knoten von den Werten $\mathrm{D}(\cdot)$ abhängig gemacht wird!

der nächste Knoten aus der Warteschlange genommen wird. Dies führt zu einer Interpretation des Ablaufs in Phasen: Man berechnet einen Kürzeste-Wege-Baum für eine Teilmenge von Knoten, beginnend mit einem Knoten. Man fügt dann einen weiteren Knoten hinzu und korrigiert den Kürzeste-Wege-Baum unter Berücksichtigung des i-ten Knotens in $O(n2^i)$ Schritten. Dies wird wiederholt, bis der Kürzeste-Wege-Baum für den kompletten Graphen berechnet ist – Gesamtaufwand $O(\sum_{i=1}^{n} n2^i) = O(n2^n)$.

3.3.3 Der Algorithmus von Pallottino

Diese Phasenaufteilung nimmt auch der Algorithmus von Pallottino vor, mit dem Unterschied, dass dieser statt des Kellers eine weitere Warteschlange benutzt. Im Algorithmus sind diese beiden Warteschlangen strikter zu trennen, als es bei der Kopplung von Keller und Warteschlange im D'Esopo-Pape-Algorithmus nötig war. Das Einfügen in die erste oder zweite Warteschlange geschieht auch hier abhängig davon, ob ein Knoten schon einen endlichen Wert hatte oder zum ersten Mal betrachtet wird. Das Entfernen erfolgt so, dass ein Knoten aus der ersten Warteschlange genommen wird, sofern diese nicht leer ist. Ist sie leer, wird ein Knoten aus der zweiten Warteschlange entnommen.

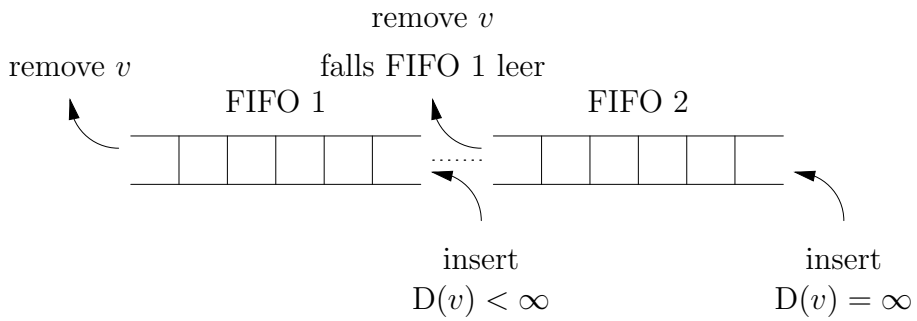


Abbildung 3.9: Die Datenstruktur im Algorithmus von Pallottino

Benötigt der D'Esopo-Pape-Algorithmus in der i-te Phase $O(n2^i)$ Schritte, so benutzt Pallottino durch die Warteschlange (FIFO 1) de facto den Bellman-Ford-Algorithmus als Unterprozedur mit Aufwand $O(im)$ in der i-ten Phase.

Satz 3.28 (Laufzeit des Pallottino-Algorithmus) *Die Laufzeit des Pallottino-Algorithmus beträgt $O(n^2m)$.*

Beweis: Die i-te Phase benötigt $O(im)$ Schritte (Bellman-Ford-Algorithmus, Satz 3.24), der Gesamtaufwand beträgt somit $O(\sum_{i=1}^{n} im) = O(n^2m)$. □

Negative Zyklen können jeweils erkannt werden, wenn der Bellman-Ford der i-ten Pallottino-Phase in der i-ten (Bellman-Ford-)Phase verkürzende Kanten entdeckt. Beim D'Esopo-Pape-Algorithmus gibt es solch ein frühzeitiges Erkennen negativer Zyklen nicht, es bleiben nur die Kriterien wie beim generischen Algorithmus.

In der Praxis ist der Pallottino-Algorithmus bei den „günstigen“ Graphen nur unwesentlich schlechter als der nach D'Esopo-Pape. Dieses ist hauptsächlich auf die etwas kompliziertere Low-Level-Programmierung zurückzuführen – in der FIFO 1 bzw. im LIFO sind dann nur selten oder nie mehr als 1 Knoten, die wiederholt bearbeitet werden müssen, der Ablauf als solcher ist dann quasi identisch. In den „schlechten“ Fällen ist der Pallottino-Algorithmus jedoch deutlich besser [CGR96].

Der Worst-Case wird bei Pallottino nur erreicht, wenn jeweils mit den schon bearbeiteten Knoten durch das Hinzufügen eines neuen Knotens wieder ein Worst-Case für den Bellman-Ford-Algorithmus abläuft (oder zumindest in einem von n unabhängigen Anteil der Phasen, wobei bei diesen auch jeweils ein von der Anzahl der betrachteten Knoten unabhängiger Anteil der Kanten betrachtet werden müsste). Ein Worst-Case-Beispiel könnte wie folgt aussehen: Der Aufbau ist ähnlich wie bei dem Worst-Case-Beispiel zum Bellman-Ford-Algorithmus nach [MN99] mit zusätzlichen Kanten vom Startknoten zu allen anderen mit ausreichend großem Gewicht. Diese Kanten sind so in der Adjazenzliste angeordnet, dass die weit vom Startknoten entfernten Knoten zuerst betrachtet werden. Dadurch wird das ursprüngliche Worst-Case-Beispiel für den Bellman-Ford-Algorithmus in jeder Phase des Pallottino-Algorithmus um einen zusätzlichen Knoten erweitert und wieder bearbeitet. Die Kantengewichte der vom Startknoten zugefügten Kanten sind vom Gewicht her so groß zu wählen, dass in der nächsten Phase des Pallottino-Algorithmus, die Werte $\mathrm{D}(\cdot)$ der betreffenden Knoten sich weiter verringern können.

3.4 Entfernungssetzende Instanzen

Bei den bisherigen Algorithmen weiß man beim Betrachten der ausgehenden Kanten eines Knotens v nicht, ob sich der Wert $\mathrm{D}(v)$ später noch einmal verringern kann und somit die von v ausgehenden Kanten später wiederholt bearbeitet werden müssen. Jede Kante uv sollte nur einmal betrachtet werden müssen. Eine konsistente Kante uv ($\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma(uv)$) kann nur dann verkürzend werden, wenn sich der Wert $\mathrm{D}(u)$ verringert.[21] Wählt man

[21] Da die Werte $\mathrm{D}(\cdot)$ stets verringert werden, ist die Alternative, dass $\mathrm{D}(v)$ erhöht wird, nicht möglich.

also den Knoten u so, dass $\mathrm{D}(u)$ bereits die kürzeste Entfernung $\mathrm{d}(u)$ angibt, so wird sich $\mathrm{D}(u)$ nicht mehr verringern können und die von u ausgehenden Kanten bleiben nach ihrer Abarbeitung konsistent. Dies führt auf ein weiteres allgemeines Schema zur Lösung des SSSP-Problems. Wir stellen hier als Sonderfälle Algorithmen für azyklische Graphen, Graphen mit $\gamma \equiv c,\ c \in \mathbb{R}$, sowie den bekannten Dijkstra-Algorithmus [Dij59] und den A^*-Algorithmus [HNR68] vor.

Wir werden in diesen Algorithmen jeden Knoten nur einmal aus der Menge R in Schritt $\mathrm{A}_{3.18}.2.1$ auswählen. Der Kürzeste-Wege-Baum wird so in jedem Schleifendurchlauf um einen Knoten erweitert. Insbesondere ist der kürzeste Weg zu einem Knoten bekannt, sobald dieser das erste Mal ausgewählt wird. Merkt man sich die Knoten, die bereits ausgewählt wurden in der Menge B, dies sind die Knoten des schon berechneten Teils des Kürzeste-Wege-Baums, so ist die Menge $R \subseteq V - B$. Genauer gilt $R = N(B) - B$ und $\max(\mathrm{D}(R)) < \infty$, für alle anderen Knoten $v \in V - (B \cup R)$ ist $\mathrm{D}(v) = \infty$. Die Verwaltung der Menge B wird hier nur zur Beschreibung und in den Beweisen benötigt und kann in der Implementierung weggelassen werden.

Für die geschätzten Entfernungen $\mathrm{D}(v)$ wird folgende wichtige Invariante stets aufrecht erhalten:

$$\mathrm{D}(v) = \min\{\mathrm{d}(u) + \gamma(uv) \mid u \in B,\ uv \in E\} \tag{3.29}$$

Mit dem Lemma 3.1 folgt, dass $\mathrm{D}(v) = \mathrm{d}(v)$, $v \in R$ gilt, wenn v auf einem beliebigen kürzesten Weg vom Startknoten zu einem beliebigen Zielknoten liegt. Diese Eigenschaft werden wir an vielen Stellen verwenden.

Für die Knoten $v \in V - (B \cup R)$ gilt die Invariante 3.29 ebenfalls, da für diese keine Kante $uv \in E$ mit $u \in B$ existiert (sonst wäre $v \in R$) und somit ist korrekterweise $\mathrm{D}(v) = \min(\emptyset) = \infty$.

Algorithmus 3.30 *Schema zur Lösung des SSSP Problems*

$\mathrm{A}_{3.30}.1$ Init_SSSP(v_0); $R := \{v_0\}$; $B := \emptyset$;
$\mathrm{A}_{3.30}.2$ **while** $\exists\ u \in R$ **do**
$\mathrm{A}_{3.30}.2.1$ wähle $u \in R$ mit $\mathrm{D}(u) = \mathrm{d}(u)$; $R := R - \{u\}$; $B := B \cup \{u\}$;
$\mathrm{A}_{3.30}.2.2$ **for all** $uv \in E$ **do**
$\mathrm{A}_{3.30}.2.2.1$ **if** is_decreasing(u, v) **then**
$\mathrm{A}_{3.30}.2.2.1.1$ decrease_key(u, v); $R := R \cup \{v\}$ **fi od od**

Wir lassen zunächst offen, wie wir in Schritt $\mathrm{A}_{3.30}.2.1$ das u wählen, sodass wir $\mathrm{D}(u) = \mathrm{d}(u)$ garantieren können. Die Terminierung und Korrektheit des

Algorithmus setzt nur voraus, dass solch ein u stets existiert. Dieses werden wir nun beweisen.

Satz 3.31 *Der Algorithmus 3.30 löst das SSSP Problem für Graphen $G = (V, E, \gamma)$ ohne negative Zyklen mit Kantengewichten $\gamma : E \to \mathbb{R}$ und Startknoten $v_0 \in V$. (Insbesondere existiert stets ein $v \in R$ mit $\mathrm{D}(v) = \mathrm{d}(v)$ solange $R \neq \emptyset$ ist.)*

Beweis: Wir beweisen Terminierung und Korrektheit durch Induktion über die Anzahl der Schleifendurchläufe: In $\mathrm{A}_{3.30}.1$ wird die Invariante 3.29 hergestellt. Sei $w = v_0 v_1 \cdots v_k$ ein beliebiger kürzester Weg von v_0 nach v_k mit $v_k \in V - B$ und v_i der erste Knoten auf diesem Weg in $V - B$ (nach Invariante 3.29 ist $v_i \in R$), so ist $w' = v_0 v_1 \cdots v_{i-1} v_i$ nach Lemma 3.1 ein kürzester Weg von v_0 nach v_i und

$$\begin{array}{rcll} \mathrm{d}(v_i) & \leq & \mathrm{D}(v_i) & \text{nach Definition } \mathrm{D}(\cdot) \\ & = & \min\{\mathrm{d}(u) + \gamma(uv_i) \mid u \in B\} & \text{Invariante 3.29} \\ & \leq & \mathrm{d}(v_{i-1}) + \gamma(v_{i-1}v_i) & \text{Minimumbildung} \\ & = & \mathrm{d}(v_i) & \text{Lemma 3.1} \end{array}$$

In $\mathrm{A}_{3.30}.2.1$ existiert also stets ein $v \in V - B$ mit $\mathrm{D}(v) = \mathrm{d}(v)$ und die Invariante 3.29 wird in $\mathrm{A}_{3.30}.2.2$ wieder hergestellt. B wird in jedem Schleifendurchlauf um einen Knoten erweitert. Kanten uv mit $v \in B$ können nicht verkürzend sein, da $\mathrm{D}(v) = \mathrm{d}(v)$ bereits gilt, jeder Knoten kann deshalb nur einmal aus R ausgewählt werden. Somit terminiert der Algorithmus 3.30, wenn alle Knoten erreicht sind, und berechnet die kürzesten Entfernungen korrekt. □

Die Laufzeit ist $O(n)$ für die Verwaltung der Knoten und $O(m)$ für die Betrachtung der ausgehenden Kanten, jeder Knoten und jede Kante wird genau einmal bearbeitet. Es verbleibt zusätzlich nur der Aufwand zur Wahl eines Knotens $u \in R$ mit $\mathrm{D}(u) = \mathrm{d}(u)$.

Wir werden nun für azyklische Graphen, Graphen mit $\gamma \equiv c,\ c \in \mathbb{R}$, für Graphen mit nichtnegativen Kantengewichten und für eine heuristische Variante zeigen, nach welchen Kriterien ein Knoten $u \in R$ effizient ausgewählt werden kann, sodass $\mathrm{D}(u) = \mathrm{d}(u)$ erfüllt ist.

3.4.1 Ein Linearzeit-Algorithmus für azyklische Graphen

In azyklischen Graphen mit beliebigen Kantengewichten lässt sich das SSSP-Problem in Linearzeit lösen. Dies beruht auf der Tatsache, dass sich in azykli-

schen Graphen die Knoten so nummerieren lassen, dass für alle Kanten v_iv_j gilt, dass $i < j$. Diese topologische Sortierung lässt sich mittels Tiefensuche in $O(n + m)$ berechnen ([CLRS01], [Tar72]). Die Menge R wird hier nicht explizit benötigt.

Satz 3.32 *Sei $G = (V, E, \gamma)$ azyklisch mit $V = \{v_0, v_1, \ldots, v_{n-1}\}$ und für alle Kanten v_iv_j gelte $i < j$, dann lässt sich das SSSP-Problem in G in $O(n + m)$ lösen.*

Beweis: O. B. d. A. sei der Startknoten v_0.[22] Wir zeigen durch Induktion, dass mit der Wahl $u = v_{i-1}$ im i-ten Schleifendurchlauf das SSSP-Problem korrekt gelöst wird – am Ende des i-ten Schleifendurchlaufs gilt für $v_j,\ j \leq i$, dass $\mathrm{D}(v_j) = \mathrm{d}(v_j)$. Im ersten Schleifendurchlauf wird v_0 mit $\mathrm{D}(v_0) = \mathrm{d}(v_0) = 0$ gewählt und die Kanten v_0v_i sind nach dem Setzen von $\mathrm{D}(v_i) := \gamma(v_0v_i)$ konsistent.

Nach Bearbeitung der von v_i ausgehenden Kanten sind alle Kanten v_jv_k, $j \leq i$ konsistent ($\mathrm{D}(v_k) \leq \mathrm{D}(v_j) + \gamma(v_jv_k)$). Da es keine Kanten $v_{j'}v_{k'}$ mit $j' \geq i$ und $k' \leq i$ gibt, können sich die Werte $\mathrm{D}(j),\ j \leq i$ nicht mehr verringern und somit die Kanten $v_jv_k,\ j \leq i$ nicht mehr verkürzend werden. Damit wird jeder Knoten und jede Kante im Algorithmus nur einmal betrachtet und die Korrektheit folgt direkt aus Lemma 3.10, da am Ende alle Kanten konsistent sind. □

Die topologische Sortierung braucht nicht vorab berechnet zu werden, sondern kann parallel zur Kürzeste-Wege-Berechnung geschehen [DF79]: Dazu wird ein Array $g[0 \ldots n-1]$ mit den Eingangsgraden der Knoten $g[i] := |\{v_jv_i \in E\}|$ initialisiert; dies gibt im weiteren Verlauf an, wieviel der in v_i eingehenden Kanten bzgl. Konsistenz noch nicht betrachtet wurden. Die Werte $\mathrm{D}(\cdot)$ der Knoten mit $g[\cdot] = 0$ können sich nicht mehr verringern und können somit als nächstes ausgewählt werden. Man verwaltet dazu eine Menge *Next* (initialisiert mit $\mathit{Next} := \{v_0\}$) und wählt im Schritt $\mathrm{A}_{3.30}.2.1$ stets einen Knoten aus *Next* – bei jeder betrachteten Kante v_iv_j wird $g[j]$ um eins verringert und v_j in *Next* eingefügt, falls $g[j] = 0$ gilt. Verbleiben zum Schluss Knoten mit $g[\cdot] > 0$, so war G nicht azyklisch. Der Gesamtaufwand bleibt offensichtlich $O(n + m)$.

In azyklischen Graphen lassen sich auch die längsten Wege in Linearzeit berechnen: Dazu wird entweder statt der Minimumbildung in $\mathrm{A}_{3.30}.2.2$ eine Maximumbildung durchgeführt und statt decrease_key$(\cdot)$ die Werte $\mathrm{D}(\cdot)$

[22] Ist der Startknoten v_i, so sind die Knoten $v_0, \ldots, v_{i-1}$ nicht erreichbar und wir betrachten das Problem auf dem von den Knoten $v_i, \ldots, v_{n-1}$ induzierten Untergraphen.

ggf. nur nach oben korrigiert oder man berechnet die kürzesten Wege in $G' = (V, E, \gamma')$ mit $\gamma'(\cdot) = -\gamma(\cdot)$ und gibt die Werte $-\mathrm{d}(\cdot)$ aus – diese entsprechen dann den längsten Wegen.

Im Allgemeinen ist die Berechnung des längsten Weges in Graphen mit Zyklen positiver Länge analog zur Berechnung der kürzesten Wege in Graphen mit negativen Zyklen nicht möglich. Die Berechnung des längsten doppelpunktfreien Weges ist NP-vollständig [GJ79].

3.4.2 Ein Linearzeit-Algorithmus für Graphen mit $\gamma \equiv c$

Der Algorithmus von Bellman-Ford läuft im allgemeinen Fall in $O(n \cdot m)$ (Satz 3.24). Wir zeigen nun, dass der Algorithmus bei konstanten Kantengewichten $\gamma(uv) = c,\ uv \in E,\ c \geq 0$ in Linearzeit läuft.[23]

Satz 3.33 *Sei $G = (V, E, \gamma)$ mit $\gamma(uv) = c,\ uv \in E,\ c \geq 0$. Der Bellman-Ford-Algorithmus löst das SSSP-Problem in G in $O(n + m)$.*

Beweis: Da alle Kanten gleiches Kantengewicht haben, so entscheidet nur die Anzahl der Kanten, sei also o. B. d. A. $c = 1$. Der Beweis ist sehr ähnlich dem Beweis des Phasen-Lemmas 3.23. Wir zeigen durch Induktion eine etwas stärkere Aussage: Für alle Knoten v gilt stets entweder $\mathrm{D}(v) = \mathrm{d}(v)$ oder $\mathrm{D}(v) = \infty$, wobei in der $(k-1)$-ten Phase die Knoten v mit $\mathrm{d}(v) = k - 1$ ihren korrekten Wert erhalten haben und genau diese sich (für die Phase k) in der Warteschlange befinden. Nach der Initalisierung („0. Phase“) gilt $\mathrm{D}(v_0) = \mathrm{d}(v_0) = 0$, $\mathrm{D}(v) = \infty$ sonst, und nur der Knoten v ist in der Warteschlange – die erste Phase kann beginnen. Gelte die Aussage für $k-1$. Für jeden Knoten v_k mit $\mathrm{d}(v_k) = k$ existiert ein kürzester Weg $w = v_0 v_1 \cdots v_{k-1} v_k$. Nach Lemma 3.1 ist $w' = v_0 v_1 \cdots v_{k-1}$ kürzester Weg nach v_{k-1}, somit $\mathrm{d}(v_{k-1}) = k - 1$. Nach Induktionsvoraussetzung befindet sich nach Beendigung der $(k-1)$-ten Phase v_{k-1} in der Warteschlange und $\mathrm{D}(v_k) = \infty$. In Phase k werden die ausgehenden Kanten von v_{k-1} betrachtet, dabei wird $\mathrm{D}(v_k)$ von ∞ auf $\mathrm{D}(v_k) := \mathrm{d}(v_{k-1}) + \gamma(v_{k-1}v_k) = \mathrm{d}(v_k)$ verringert und v_k befindet sich somit am Ende der k-ten Phase in der Warteschlange.

[23] Ist c negativ, so sind die kürzesten Wege nur in azyklischen Graphen wohldefiniert. Es kann der Algorithmus für azyklische Graphen verwendet werden. Falls bei der topologischen Sortierung Zyklen entdeckt werden, so sind die kürzesten Wege durch die negativen Kantengewichte nicht wohldefiniert.

Da die Werte $\mathrm{D}(\cdot)$ stets von ∞ direkt auf den korrekten Wert $\mathrm{d}(\cdot)$ gesetzt werden, kann jeder Knoten sich nur einmal in der Warteschlange befinden und somit wird jede Kante nur einmal betrachtet. Die Laufzeit ist $O(n+m)$. □

3.4.3 Der Dijkstra-Algorithmus für Graphen mit $\gamma(\cdot) \geq 0$

Dijkstras Algorithmus ([Dij59]) wählt stets den Knoten aus R, der die minimale obere Schranke $\mathrm{D}(\cdot)$ annimmt. Wir zeigen, dass für diesen Knoten $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ gilt.

Lemma 3.34 ($\mathrm{D}(v) = \mathrm{d}(v)$ nach Dijkstra) *Sei $G = (V, E, \gamma)$ mit $\gamma(\cdot) \geq 0$. Sei $v \in V - B$ mit $\mathrm{D}(v) = \min(\mathrm{D}(V - B))$, so gilt $\mathrm{D}(v) = \mathrm{d}(v)$.*

Beweis: Sei mit $v_k = v$ der Weg $w = v_0 v_1 \cdots v_k$ ein beliebiger kürzester Weg von v_0 nach v und v_i der erste Knoten auf diesem Weg in $V - B$, so gilt

$$\begin{array}{rcll} \mathrm{D}(v) & \geq & \mathrm{d}(v) & \text{nach Definition } \mathrm{D}(\cdot) \\ & \geq & \mathrm{d}(v_i) & \text{nach Lemma 3.1 mit } \gamma(\cdot) \geq 0 \\ & = & \mathrm{D}(v_i) & \text{Invariante 3.29} \\ & \geq & \mathrm{D}(v) & \text{Auswahl nach Minimumbildung} \end{array}$$

und somit also $\mathrm{D}(v) = \mathrm{d}(v)$.[24] □

Gilt für alle uv, dass $\gamma(uv) > 0$, so ist im Beweis zu Lemma 3.34 stets $v_i = v_k$.

Eine wichtige Eigenschaft des Lemmas 3.34 ist die Reihenfolge, in der die Knoten in die Menge B aufgenommen werden.

Lemma 3.35 *Im Algorithmus 3.30 implementiert mit Lemma 3.34 werden die Knoten in aufsteigender Reihenfolge der Entfernung zum Startknoten in die Menge B aufgenommen.*

Beweis: Wird ein Knoten u mit $\mathrm{D}(u) = \min(\mathrm{D}(V - B))$ ausgewählt, so gilt mit Lemma 3.34 $\mathrm{D}(u) = \mathrm{d}(u)$. Für die Knoten $v \in V - B$, deren Wert $\mathrm{D}(v)$ in $\mathrm{A}_{3.30}.2.2$ nicht verändert wurde, gilt $\mathrm{D}(u) = \min(\mathrm{D}(\{u\} \cup (V - B))) \leq \mathrm{D}(v)$, ansonsten wurde $\mathrm{D}(v) := \mathrm{D}(u) + \gamma(uv) \geq \mathrm{D}(u)$ gesetzt. Das in der nächsten Iteration gewählte u' kann also keinen Wert $\mathrm{D}(u') < \mathrm{D}(u) = \mathrm{d}(u)$ haben. □

[24] Beachte, dass hier Kantengewichte $\gamma(\cdot) = 0$ die Korrektheit des Beweises nicht beeinflussen. Bei dem Standardbeweis werden Kantengewichte > 0 vorausgesetzt.

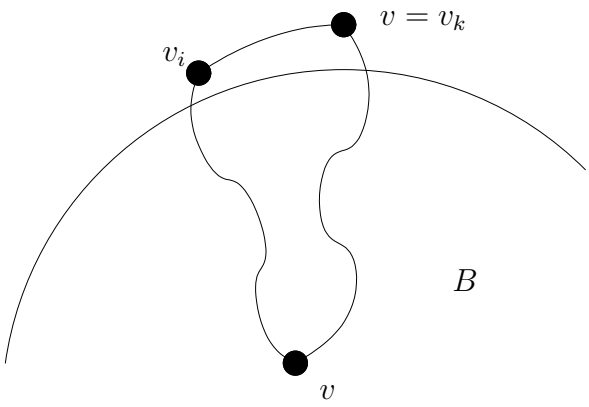


Abbildung 3.10: Illustration des Beweises zu Lemma 3.34

Nach [Tho99] formuliert lautet das Lemma 3.35: Im Algorithmus 3.30 implementiert mit Lemma 3.34 gilt stets $\min(\mathrm{D}(V-B)) = \min(\mathrm{d}(V-B))$, was sich auch analog zum Beweis von Lemma 3.34 zeigen lässt. Der Knoten v_i sei der erste nicht in B liegende Knoten auf einem kürzesten Weg nach v mit $\mathrm{d}(v) = \min(\mathrm{d}(V-B))$:

$$\min(\mathrm{D}(V-B)) \geq \min(\mathrm{d}(V-B)) = \mathrm{d}(v) \geq \mathrm{d}(v_i) = \mathrm{D}(v_i) \geq \min(\mathrm{D}(V-B))$$

Die Reihenfolge der aufsteigenden Entfernungen folgt, da die $\mathrm{d}(\cdot)$-Werte konstant sind und das Minimum über eine immer kleiner werdende Menge $V-B$ gebildet wird.

Korollar 3.36 *Bei Verwendung von Lemma 3.34 gilt* $\mathrm{d}(u) \leq \mathrm{d}(v)$, $\forall u \in B, v \in V-B$.

Im Algorithmus 3.30 implementiert mit Lemma 3.34 erfolgt die Auswahl der Knoten erstmals aufgrund der Werte $\mathrm{D}(\cdot)$. Abstrakt betrachtet werden die Werte $\mathrm{D}(R)$ in einer Datenstruktur verwaltet, die die folgenden Operationen unterstützt:

- insert(v, x) fügt den Knoten v mit Wert $\mathrm{D}(v) := x$ in die Menge R ein.
- decrease_key(u, v) verringert für den Knoten v den Wert $\mathrm{D}(v)$ auf $\mathrm{D}(u) + \gamma(uv)$, d. h., $\mathrm{D}(u) + \gamma(uv)$ muss kleiner als der alte Wert $\mathrm{D}(v)$ sein.
- delete_min(R) gibt einen Knoten v mit Wert $\mathrm{D}(v) := \min(\mathrm{D}(R))$ zurück und entfernt ihn aus R.

Die Auswahl eines Knotens v mit $\mathrm{D}(v) = \mathrm{d}(v)$ und das Entfernen aus R entspricht dann der Zuweisung $v :=$ delete_min(R).

Satz 3.37 *Die Laufzeit des Algorithmus 3.30 implementiert mit Lemma 3.34 beträgt $O(n \cdot \log n + m)$.*

Beweis: Die Laufzeit des Algorithmus 3.30 implementiert mit Lemma 3.34 ist bestimmt durch $n-1$ Aufrufe von delete_min(R) zur Bestimmung der $v \in V-B$ mit $\mathrm{D}(v) = \min(\mathrm{D}(V-B))$ sowie maximal m Aufrufe von insert$(\cdot)$ und decrease_key$(\cdot)$ in den Schritten $\mathrm{A}_{3.30}.1$ bzw. $\mathrm{A}_{3.30}.2.2$. Implementiert man die Prozeduren insert$(\cdot)$, decrease_key$(\cdot)$ und delete_min(R) mit Fibonacci Heaps (Anhang A, [FT87]) ergibt sich ein amortisierter Aufwand von je $O(\log n)$ für die Auswahl der Knoten und $O(1)$ je Kante in den Aktualisierungsschritten, somit beträgt die Gesamtlaufzeit $O(n \cdot \log n + m)$.[25] □

Satz 3.38 *Jede vergleichsbasierte Implementierung des Dijkstra-Algorithmus hat im Worst-Case eine Laufzeit von $\Omega(m + n \cdot \log n)$.*

Beweis: Aus dem Lemma 3.35 und dem Korollar 3.36 folgt direkt, dass man n natürliche Zahlen $z_1, z_2, \ldots, z_n$ dadurch sortieren kann, dass man im Graphen $G = (V, E, \gamma)$ mit $V = \{v_0, \ldots, v_n\}$, $E = \{v_0 v_i | 0 < i \leq n\}$ und $\gamma(v_0 v_i) = z_i$ den Dijkstra Algorithmus mit Startknoten v_0 ablaufen lässt. Die Reihenfolge, in der die Knoten in B aufgenommen werden, entspricht der aufsteigenden Sortierung der Zahlen. Dies heißt insbesondere, dass in einem vergleichsbasierten Modell die Laufzeit im Worst-Case einerseits $\Omega(n \cdot \log n)$ beträgt, andererseits kann jede Kante zu einem kürzesten Weg gehören, wodurch die Laufzeit $\Omega(m)$ betragen muss. Somit ist die Implementierung des Dijkstra-Algorithmus mit Fibonacci Heaps bzgl. O-Notation optimal. □

Dies heißt insbesondere auch, dass alle Verbesserungen der asymptotischen Laufzeit entweder die Knoten nicht in aufsteigender Reihenfolge der Entfernungen aufnehmen (müssen) oder nicht ausschließlich auf Vergleichen basieren.

[25] Mit einer Listenimplementierung ergibt sich $O(1)$ für insert$(\cdot)$ und decrease_key$(\cdot)$, sowie $O(n)$ für delete_min(R), somit ein Gesamtaufwand $O(n^2)$. Mit Heaps ergibt sich je $O(\log n)$ für jede der drei Operationen, somit $O(m \cdot \log n)$, was für Graphen mit $m \in O(n^2 / \log n)$ Kanten besser als die Listenimplementierung ist, gegenüber der Implementierung mit Fibonacci-Heaps bei Graphen mit $m \in \omega(n \cdot \log n)$ jedoch um einen Faktor $\log n$ schlechter.

3.4.4 Der A^*-Algorithmus, eine heuristische Variante

Der Algorithmus 3.30 mit Lemma 3.34 berechnet die Entfernung vom Startknoten zu allen anderen Knoten und findet dabei die kürzesten Wege in wachsendem Abstand zum Startknoten. Möchte man nur die Entfernung zu einem bestimmten Zielknoten berechnen (SPSP-Problem), so würde man von Hand z. B. auf einer Landkarte zielgerichtet vorgehen und vorzugsweise Wege in Zielrichtung betrachten. Liegen Informationen für alle Knoten vor, wie weit diese höchstens noch vom Ziel entfernt sind, so kann man zur Auswahl des Knotens im Schritt $A_{3.30}.2.1$ alternativ das Lemma 3.40 verwenden. Wir benötigen vorab noch folgende Definition:

Definition 3.39 (Zulässige und konsistente Schätzfunktionen) *Zu einem Zielknoten $z \in V$ heißt* $\widehat{\mathrm{ed}}_z : V \to \mathbb{R}$ Schätzfunktion. $\widehat{\mathrm{ed}}_z : V \to \mathbb{R}$ *heißt* zulässig *genau dann, wenn* $\widehat{\mathrm{ed}}_z(z) = 0$ *und* $\widehat{\mathrm{ed}}_z(u) \leq \mathrm{dist}(u, z),\ u \in V$. $\widehat{\mathrm{ed}}_z : V \to \mathbb{R}$ *heißt* konsistent *genau dann, wenn* $\widehat{\mathrm{ed}}_z(z) = 0$ *und* $\widehat{\mathrm{ed}}_z(u) \leq \mathrm{dist}(u, v) + \widehat{\mathrm{ed}}_z(v),\ u, v \in V$.

[Pea84] schlägt vor, die effizienter zu überprüfende Bedingung $\widehat{\mathrm{ed}}_z(z) = 0$ und $\widehat{\mathrm{ed}}_z(u) \leq \gamma(uv) + \widehat{\mathrm{ed}}_z(v),\ uv \in E$ zu verwenden (dort *Monotonie* genannt). Mit $\mathrm{dist}(u, v) \leq \gamma(uv)$ folgt aus Konsistenz die Monotonie, für beliebige Wege $w = v_0 v_1 \cdots v_k$ gilt mit der Monotonie

$$\widehat{\mathrm{ed}}_z(v_0) \leq \gamma(v_0 v_1) + \widehat{\mathrm{ed}}_z(v_1) \leq \cdots \leq \gamma(v_0 v_1 \cdots v_k) + \widehat{\mathrm{ed}}_z(v_k)$$

Dies gilt insbesondere auch für kürzeste Wege, somit folgt die Konsistenz.

Wir werden den Zusammenhang zwischen dem Begriff der Konsistenz bei Kanten ($\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma(uv),\ uv \in E$) und der Konsistenz bei Schätzfunktionen im Folgeabschnitt 3.5 noch untersuchen, betrachten hier aber zunächst, wie Schätzfunktionen bei der Kürzeste-Wege-Suche eingesetzt werden können.

Lemma 3.40 ($\mathrm{D}(v) = \mathrm{d}(v)$ nach A^*) *Sei $z \in V$ der Zielknoten und $\widehat{\mathrm{ed}}_z : V \to \mathbb{R}$ eine konsistente Schätzfunktion. Sei $v \in V - B$ mit $\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) = \min_{v' \in V-B}\{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\}$, so gilt $\mathrm{D}(v) = \mathrm{d}(v)$.*

Beweis: Angenommen v sei der erste Knoten, der die Bedingung $\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) = \min_{v' \in V-B}\{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\}$ trotz $\mathrm{D}(v) > \mathrm{d}(v)$ erfüllt. Dann existiert ein Weg w nach v mit $\gamma(w) = \mathrm{d}(v) < \mathrm{D}(v)$, $w = v_0 \cdots v_i \cdots v$, und v_i sei der Knoten in $V - B$ mit i minimal – es ist $v_i \neq v$, da sonst wegen der

Invariante 3.29 $\mathrm{D}(v) = \mathrm{d}(v)$ gelten würde. Der Weg $v_0 \cdots v_i$ ist als Teilweg eines kürzesten Weges nach v nach Lemma 3.1 kürzester Weg nach v_i. Es gilt

$$\begin{array}{rcll} \mathrm{D}(v_i) + \widehat{\mathrm{ed}}_z(v_i) & = & \mathrm{d}(v_i) + \widehat{\mathrm{ed}}_z(v_i) & \text{Invariante 3.29} \\ & \leq & \mathrm{d}(v_i) + \mathrm{dist}(v_i, v) + \widehat{\mathrm{ed}}_z(v) & \text{Konsistenz von } \widehat{\mathrm{ed}}_z(\cdot) \\ & = & \mathrm{d}(v) + \widehat{\mathrm{ed}}_z(v) & \text{Lemma 3.1} \\ & < & \mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) & \text{Annahme } \mathrm{D}(v) > \mathrm{d}(v) \end{array}$$

im Widerspruch zur Annahme $\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) = \min_{v' \in V-B}\{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\}$. □

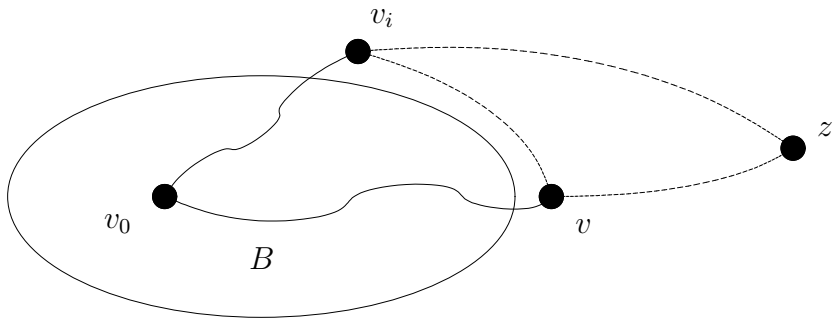


Abbildung 3.11: Illustration des Beweises zu Lemma 3.40

Werden in R durch insert$(\cdot)$ bzw. decrease_key$(\cdot)$ die Werte $\mathrm{D}(\cdot) + \widehat{\mathrm{ed}}_z(\cdot)$ verwaltet, so entspricht die Auswahl eines Knotens v nach Lemma 3.40 auch hier $v :=$ delete_min(R).

Während im Worst-Case die Laufzeit weiterhin $O(m + n \cdot \log n)$ beträgt, geben [SV86] für den Fall von Euklidischen Graphen (mit $\gamma(uv) = \mathrm{d}^{(e)}(u, v)$, $uv \in E$, und $\widehat{\mathrm{ed}}_z(v) = \mathrm{d}^{(e)}(v, z),\ v, z \in V$) folgende Abschätzung für den Average-Case (hier ohne Beweis).

Satz 3.41 *Die Laufzeit des Algorithmus 3.30 implementiert mit Lemma 3.40 beträgt für Euklidische Graphen im Average-Case $O(n)$.*

Dies ist insofern sehr bemerkenswert, da die Anzahl der Kanten $O(n^2)$ betragen kann.

Analog zum Lemma 3.35 gilt eine ähnliche Eigenschaft auch für den A^*-Algorithmus:

Lemma 3.42 *Im Algorithmus 3.30 implementiert mit Lemma 3.40 werden die Knoten in aufsteigender Reihenfolge der Werte $\mathrm{d}(\cdot) + \widehat{\mathrm{ed}}_z(\cdot)$ in die Menge B aufgenommen.*

Beweis: Wird ein Knoten u mit

$$\mathrm{D}(u) + \widehat{\mathrm{ed}}_z(u) = \min_{v' \in V-B} \{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\}$$

ausgewählt, so gilt mit Lemma 3.40 $\mathrm{D}(u) = \mathrm{d}(u)$. Für die Knoten v, deren Wert $\mathrm{D}(v)$ in $\mathrm{A}_{3.30}$.2.2 nicht verändert wurde, gilt

$$\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) \geq \min_{v' \in \{u\} \cup (V-B)} \{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\} = \mathrm{D}(u) + \widehat{\mathrm{ed}}_z(u)$$

Für die anderen Knoten v wurde $\mathrm{D}(v) := \mathrm{D}(u) + \gamma(uv)$ gesetzt, somit

$$\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) = \mathrm{D}(u) + \gamma(uv) + \widehat{\mathrm{ed}}_z(v) \geq \mathrm{D}(u) + \widehat{\mathrm{ed}}_z(u)$$

aufgrund der Konsistenzeigenschaft der Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$. Für das in der nächsten Iteration gewählte u' kann demzufolge nicht $\mathrm{D}(u') + \widehat{\mathrm{ed}}_z(u') < \mathrm{D}(u) + \widehat{\mathrm{ed}}_z(u)$ gelten. □

Wie beim Dijkstra-Algorithmus lässt sich auch diese Aussage elegant umformulieren: Es gilt im Algorithmus 3.30 implementiert mit Lemma 3.40 stets $\min_{v \in V-B_s}\{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\} = \min_{v \in V-B_s}\{\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v)\}$. Auch hier folgt die aufsteigende Reihenfolge, da die $\mathrm{d}(\cdot)$-Werte konstant sind und das Minimum über eine immer kleiner werdende Menge $V - B$ gebildet wird.

Soll nur der kürzeste Weg zum Zielknoten berechnet werden, so kann man abbrechen, sobald der Zielknoten z aus R gewählt wurde. Das Lemma 3.40 gilt aber auch darüber hinaus. Der Algorithmus 3.30 implementiert mit Lemma 3.40 ist also auch geeignet, um das SSSP-Problem zu lösen. Dabei werden die korrekten Entfernungen in Richtung Zielknoten, wie im Lemma 3.42 gezeigt, zuerst bestimmt.

3.5 Weitere Eigenschaften des Dijkstra- und A^*-Algorithmus

Der Zusammenhang zwischen dem Dijkstra- und A^*-Algorithmus sind weitergehend, als die Ähnlichkeiten im Programm es auf den ersten Blick erkennen lassen. Offensichtlich sind die folgenden zwei Eigenschaften:

Lemma 3.43 *Sind die Kantengewichte $\gamma(\cdot) \geq 0$ positiv, so ist die Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot) \equiv 0$ konsistent und der A^*-Algorithmus verhält sich identisch dem Dijkstra-Algorithmus (Lemma 3.34).*

Beachte: Sind auch negative Kantengewichte zugelassen, so ist $\widehat{\mathrm{ed}}_z(\cdot) \equiv 0$ nicht konsistent.

Lemma 3.44 *Mit der konsistenten Schätzfunktion* $\widehat{\mathrm{ed}}_z(\cdot) \equiv \mathrm{dist}(\cdot, z)$ *kann man die Knoten eines kürzesten Weges* w *von* v_0 *nach* z *der Reihe nach in* B *aufnehmen.*

Beweis: Für alle Knoten v, die nicht auf einem kürzesten Weg von v_0 nach z liegen, ist $\mathrm{d}(v) + \widehat{\mathrm{ed}}_z(v) = \mathrm{d}(v) + \mathrm{dist}(v, z) > \mathrm{d}(z)$ und so werden die Knoten v auf einem kürzesten Weg vorher aufgenommen – für diese ist $\mathrm{d}(v) + \widehat{\mathrm{ed}}_z(v) = \mathrm{d}(z)$. □

Die Laufzeit muss dadurch nicht zwangsweise besser als im Worst-Case sein. Zum Einen könnten alle Knoten auf einem kürzesten Weg liegen. Zum Anderen ist der kürzeste Weg i. A. nicht eindeutig (z. B. in Gittergraphen), sodass bei der Auswahl das nächsten Knotens v (für den wieder $\mathrm{d}(v)+\widehat{\mathrm{ed}}_z(v) = \mathrm{d}(z)$ gelten wird) zu beachten ist, dass dieser den schon begonnenen kürzesten Weg fortsetzt – ohne diese Änderung des Algorithmus könnten sonst u. U. alle Knoten vor dem Zielknoten zunächst in R und dann in B aufgenommen werden.

Im Beweis zu Lemma 3.40 wird im Gegensatz zum Lemma 3.34 nicht vorausgesetzt, dass die Kantengewichte positiv sind. Der A^*-Algorithmus funktioniert auch mit negativen Kantengewichten, sofern die Schätzfunktion konsistent ist. Wir wollen diesen Umstand nun noch einmal genauer untersuchen.

Wir vergleichen den A^*-Algorithmus mit (konsistenter) Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ und den Dijkstra-Algorithmus mit Potentialfunktion $\pi(\cdot) = -\widehat{\mathrm{ed}}_z(\cdot)$. Im A^*-Algorithmus wird der Knoten $v \in V - B$ ausgewählt für den $\mathrm{D}(\cdot) + \widehat{\mathrm{ed}}_z(\cdot)$ minimal ist. Im Dijkstra-Algorithmus wird auf den reduzierten Kantengewichten $\gamma_\pi(uv) = \pi(u)+\gamma(uv)-\pi(v) = -\widehat{\mathrm{ed}}_z(u)+\gamma(uv)+\widehat{\mathrm{ed}}_z(v)$ gearbeitet. Für die reduzierten Kantengewichte gilt $\gamma_\pi(uv) \geq 0$, da wegen der Konsistenz der Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ gilt, dass $\widehat{\mathrm{ed}}_z(u) \leq \gamma(uv) + \widehat{\mathrm{ed}}_z(v)$.[26] Ein Knoten v_k, der seinen Wert über den Weg $w_k = v_0 \cdots v_k$ erhalten hat, hat im A^*-Algorithmus den Wert $\mathrm{D}(v_k) = \gamma(w_k)$, im Dijkstra-Algorithmus beträgt der Wert $\mathrm{D}(v_k) = \gamma_\pi(w_k) = -\widehat{\mathrm{ed}}_z(v_0) + \gamma(w_k) + \widehat{\mathrm{ed}}_z(v_k)$. Die $\mathrm{D}(\cdot)$-Werte im

[26] Man beachte, dass die Konsistenz der Schätzfunktion $\widehat{\mathrm{ed}}_z(u) \leq \gamma(uv)+\widehat{\mathrm{ed}}_z(v)$ und die Konsistenz von Kanten $\mathrm{D}(v) \leq \mathrm{D}(u) + \gamma(uv)$ auf den ersten Blick gegensätzlich definiert zu sein scheinen. Berücksichtigt man, dass die Konsistenz der Kanten vom Startknoten aus betrachtet wird und die Konsistenz der Schätzfunktion vom Zielknoten aus, so löst sich dieser scheinbare Konflikt auf.

Dijkstra-Algorithmus und die Werte $\mathrm{D}(\cdot) + \widehat{\mathrm{ed}}_z(\cdot)$ im A^*-Algorithmus unterscheiden sich also nur durch die Konstante(!) $\widehat{\mathrm{ed}}_z(v_0)$. Da sich alle Werte gleichermaßen um diese Konstante unterscheiden, spielt diese bei der Auswahl der Knoten über die Minimumbildung keine Rolle. Der A^*-Algorithmus mit (konsistenter) Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ und der Dijkstra-Algorithmus mit Potentialfunktion $\pi(\cdot) = -\widehat{\mathrm{ed}}_z(\cdot)$ verhalten sich identisch!

Insbesondere ist der A^*-Algorithmus mit negativen Kantengewichten korrekt, solange die Schätzfunktion konsistent ist.

Aus der Konsistenzbedingung $\widehat{\mathrm{ed}}_z(z) = 0$ und $\widehat{\mathrm{ed}}_z(u) \leq \mathrm{dist}(u, v) + \widehat{\mathrm{ed}}_z(v)$, $u, v \in V$ folgt mit der Wahl $v = z$ sofort die Zulässigkeitsbedingung $\widehat{\mathrm{ed}}_z(z) = 0$ und $\widehat{\mathrm{ed}}_z(u) \leq \mathrm{dist}(u, z),\ u \in V$. In [Buc00] und [Sch00a] wird der A^*-Algorithmus mit dieser schwächeren Bedingung angegeben. Die Beweise, dass für die aus R gewählten Knoten v auch dann $\mathrm{D}(v) = \mathrm{d}(v)$ gilt, sind jedoch fehlerhaft, wie auch am Gegenbeispiel in Abbildung 3.12 mit nichtkonsistenter Schätzfunktion zu sehen ist.

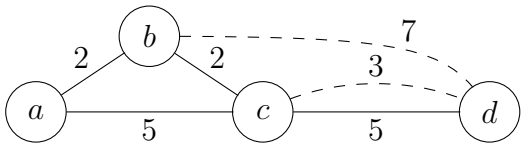


Abbildung 3.12: Beispiel für A^* mit zulässiger, aber nicht konsistenter Schätzfunktion

Die Schätzentfernungen zum Zielknoten sind hier mit gestrichelten Kanten gekennzeichnet. Wird die kürzeste Entfernung von a nach d berechnet mit $\widehat{\mathrm{ed}}_d(b) = 7$ und $\widehat{\mathrm{ed}}_d(c) = 3$, so wird der Knoten c fehlerhaft mit der Entfernung 5 angenommen. Mit $\widehat{\mathrm{ed}}_d(b) = 5$ wäre die Funktion konsistent, und es würde zunächst b, c und dann d jeweils mit den korrekten kürzesten Entfernungen in B aufgenommen werden.

Es gibt diverse Varianten des A^*-Algorithmus (siehe das umfassende Buch [Pea84] sowie [DP85], [Mér81], [BM83]), unter anderem auch eine, die mit nur zulässigen – aber nicht notwendigerweise konsistenten – Schätzfunktionen auskommt. Dort müssen aber – solange der Zielknoten noch nicht in B aufgenommen wurde – die Nachbarn eines Knotens v nochmals betrachtet werden, wenn im Verlauf der Berechnung die bisher als korrekt erachtete Entfernung $\mathrm{D}(v)$ nach unten korrigiert werden musste. Dies entspricht einem Zurückfallen auf den modifizierten generischen Algorithmus 3.18, wobei dort der nächste Knoten aufgrund der Minimumbildung über die Randknoten gewählt wird. Im Gegensatz zum Lemma 3.40 ist bei nur zulässiger Schätz-

funktion aber nicht mehr garantiert, dass für den gewählten Knoten v die Entfernung $\mathrm{D}(v) = \mathrm{d}(v)$ korrekt ist. Somit wird der Wert $\mathrm{D}(v)$ ggf. später nochmals nach unten korrigiert. So kann v auch mehrmals in R aufgenommen und ausgewählt werden.

Im Beispiel in Abbildung 3.12 wird so zunächst der Knoten c aus R gewählt, dann Knoten b. Dieser korrigiert den Wert $\mathrm{D}(c)$, sodass danach c erneut ausgewählt wird und schließlich auch d mit dem korrekten Wert $\mathrm{D}(d) = 9$ aus R ausgewählt werden kann. Alle Kanten sind nun konsistent und die Entfernungen korrekt berechnet. Die Korrektheit des A^*-Algorithmus mit nur zulässiger Schätzfunktion folgt als Sonderfall aus der Korrektheit des modifizierten generischen Algorithmus mit Satz 3.19.

Da offensichtlich der Wert $\mathrm{D}(v)$ bei erstmaliger Wahl des Knotens $v \in R$ nicht notwendigerweise korrekt ist, braucht diese Variante des A^*-Algorithmus noch eine Daseinsberechtigung: Wir zeigen nun, dass auch bei nur zulässiger Schätzfunktion der Zielknoten z nur mit $\mathrm{D}(z) = \mathrm{d}(z)$ aus R ausgewählt werden kann.

Lemma 3.45 *Im Algorithmus 3.30 mit zulässiger Schätzfunktion* $\widehat{\mathrm{ed}}_z(\cdot)$ *wird der Zielknoten z nur einmal mit dem korrekten Wert* $\mathrm{D}(z) = \mathrm{d}(z)$ *aus R ausgewählt.*

Beweis: Wir zeigen, dass, solange z noch nicht aus R gewählt wurde, stets ein Knoten $v \in R$ existiert mit $\mathrm{D}(v) + \widehat{\mathrm{ed}}_z(v) \leq \mathrm{d}(z)$, sodass z nie mit $\mathrm{D}(z) > \mathrm{d}(z)$ aus R gewählt werden kann, da dann

$$\mathrm{D}(z) + \widehat{\mathrm{ed}}_z(z) = \mathrm{D}(z) > \mathrm{d}(z) \geq \min_{v' \in R}\{\mathrm{D}(v') + \widehat{\mathrm{ed}}_z(v')\}$$

gelten würde. Sei $w = v_0 \cdots v_k$, $v_k = z$ ein kürzester Weg von v_0 nach z und v_i der erste Knoten mit $\mathrm{D}(v_i) > \mathrm{d}(v_i)$, so gilt $\mathrm{D}(v_{i-1}) = \mathrm{d}(v_{i-1})$ und v_{i-1} wurde noch nicht (mit diesem Wert $\mathrm{D}(v_{i-1})$) aus R gewählt, da sonst nach Invariante 3.29 $\mathrm{D}(v_i) = \mathrm{d}(v_i)$ gelten würde. Es gilt

$$\mathrm{D}(v_{i-1}) + \widehat{\mathrm{ed}}_z(v_{i-1}) = \mathrm{d}(v_{i-1}) + \widehat{\mathrm{ed}}_z(v_{i-1}) \leq \mathrm{d}(v_{i-1}) + \mathrm{dist}(v_{i-1}, z) = \mathrm{d}(z)$$

sodass v_{i-1} vor z aus R gewählt wird. □

Dies heißt, dass zur Berechnung des SPSP-Problems der A^*-Algorithmus (sinnvoll) zum Einsatz kommen kann, auch wenn eine nur zulässige Schätzfunktion bekannt ist. Der Algorithmus kann im Gegensatz zum generischen Algorithmus abgebrochen werden, sobald z zum ersten Mal aus R gewählt wurde.

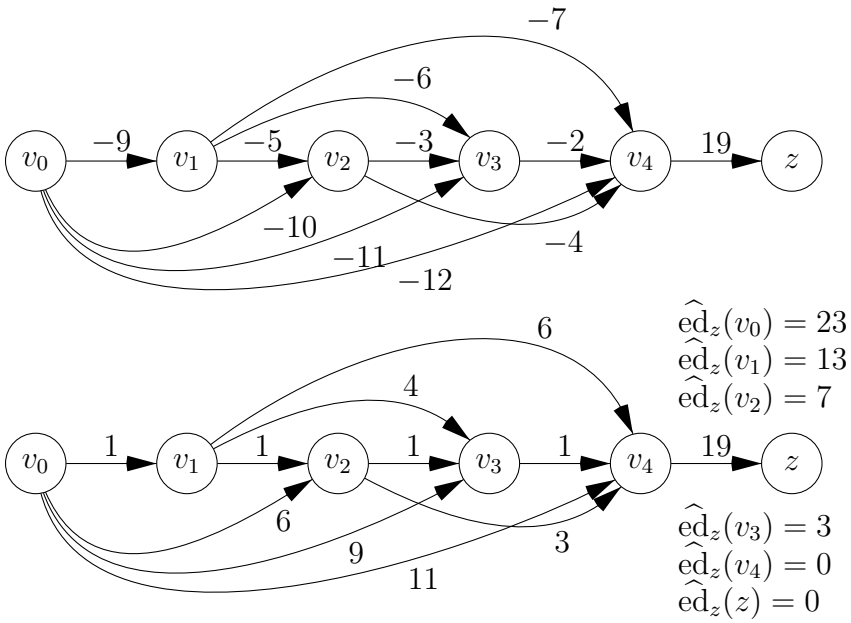


Abbildung 3.13: Beispiele für Dijkstra mit negativen Kantengewichten und modifizierten A^* mit zulässiger, aber nicht konsistenter Schätzfunktion

Wir werden nun allerdings zeigen, dass der Worst-Case $\Omega(2^n)$ Iterationen benötigt.

Zunächst kommen wir dazu auf den Zusammenhang zwischen Dijkstra- und A^*-Algorithmus zurück. Ist die Schätzfunktion nur zulässig, nicht aber konsistent, so folgt für die bzgl. der Schätzfunktion nicht konsistenten Kanten, dass die reduzierten Kantengewichte negativ sind. Beim Dijkstra-Algorithmus mit negativen Kantengewichten muss wie beim A^*-Algorithmus mit nur zulässiger Schätzfunktion ein Knoten wieder in die Menge R aufgenommen werden, wenn sich der zugehörige $\mathrm{D}(\cdot)$-Wert später nochmals verringert.[27] Führt man den Dijkstra-Algorithmus auf dem Worst-Case-Beispiel des D'Esopo-Pape-Algorithmus aus Abbildung 3.6 aus, so bearbeitet dieser die Knoten in der exakt gleichen Reihenfolge wie der D'Esopo-Pape-Algorithmus, es werden also $\Theta(2^n)$ Iterationen durchgeführt. Um eine zulässige Schätzfunktion angeben zu können, fügen wir noch einen neuen Knoten, den Zielknoten z, hinzu und verringern das Beispiel der Übersichtlichkeit halber um einen Knoten. Der zugehörige Graph ist in Abbildung 3.13 gezeigt.

[27] [EK72] führt die Modifizierung im Rahmen von Flussproblemen für den Dijkstra-Algorithmus erstmals explizit aus, [Joh73] gibt als erster ein Beispiel (ein anderes als die hier genannten), für die der so modifizierte Algorithmus exponentielle Laufzeit $O(n \cdot 2^n)$ hat.

Der obere Graph entspricht dem Beispiel aus Abbildung 3.6. Die Kantengewichte sind $\gamma(v_iv_j) = -2^{3-i} + i - j$ und $\gamma(v_4z) = -\sum_{i=0}^{3} \gamma(v_iv_{i+1}) = 19$. Der Dijkstra-Algorithmus benötigt 2^4+1 Iterationen. Mit der Potentialfunktion $\pi(v_i) := -\widehat{\mathrm{ed}}_z(v_i) = -2^{4-i} - 2(4-i) + 1$ geht das Beispiel in den unteren Graphen über. Die Schätzfunktion ist zulässig, aber nicht konsistent. Der Ablauf ist identisch dem des Dijkstra-Algorithmus im oberen Graphen und der Knoten z hat, wie im Lemma 3.45 bewiesen, bereits bei der ersten Auswahl aus R den korrekten Wert $\mathrm{D}(v) = \mathrm{d}(v)$.

Wir fassen die in diesem Abschnitt gewonnenen Erkenntnisse in folgendem Satz zusammen:

Satz 3.46 *Der A^*-Algorithmus mit konsistenter Schätzfunktion ist äquivalent zum Dijkstra-Algorithmus mit positiven Kantengewichten. Der A^*-Algorithmus mit zulässiger, aber nicht konsistenter Schätzfunktion ist äquivalent zum Dijkstra-Algorithmus, bei dem auch negative Kantengewichte erlaubt sind.*

Trotz des Worst-Cases von $O(2^n)$ Iterationen ist der A^*-Algorithmus optimal bzgl. folgendem Kriterium (hier ohne Beweis).

Satz 3.47 (Optimalität des A^*-Algorithmus, [Gel77]) *Es kann keinen heuristischen Algorithmus geben, der mit denselben Informationen wie der A^*-Algorithmus weniger Knoten als dieser bei der Lösung des SPSP-Problems betrachtet.*

Leider sagt dieser Satz nichts darüber aus, wie oft diese einzelnen Knoten angeschaut werden müssen. Er besagt lediglich, dass der A^*-Algorithmus nie Knoten betrachtet, die aufgrund der verfügbaren Informationen nicht zu einem kürzesten Weg führen können.

Zum Abschluss des Kapitels 3 sei noch angemerkt, dass das Lemma 3.27 offensichtlich für alle im Abschnitt 3.4 über entfernungssetzende Instanzen vorgestellten Algorithmen gilt, d. h., der Wert $\mathrm{D}(v)$ des ausgewählten Knotens entspricht jeweils dem durch die Vorgänger $\mathrm{pred}(\cdot)$ beschriebenen Weg. Das Lemma gilt aber auch für die entfernungskorrigierenden Varianten des Dijkstra-Algorithmus mit negativen Kantengewichten und dem A^*-Algorithmus mit zulässiger aber nicht notwendig konsistenter Schätzfunktion. Der Beweis ist analog zum Beweis des Lemmas 3.27 mit dem Unterschied, dass über die Weglänge argumentiert werden muss.

Kapitel 4

Beidseitige Suchalgorithmen

Während der A^*-Algorithmus meist zur Lösung des SPSP-Problems benutzt wird, löst er auch das SSSP-Problem, wenn man nicht nach der Berechnung der kürzesten Entfernung zum Zielknoten abbricht. Die hier nun vorgestellten beidseitigen Suchalgorithmen benutzen zwar die SSSP-Algorithmen als Komponenten, lösen aber ausschließlich das SPSP-Problem.

4.1 Einfache beidseitige Suchalgorithmen

Der Algorithmus 3.30 (z. B. in Form des Dijkstra- oder A^*-Algorithmus) kann leicht umgeformt werden, um von allen Knoten durch Rückwärtssuche die kürzesten Entfernungen zu einem Zielknoten zu bestimmen (SDSP-Problem). Die Vorwärts- und Rückwärtssuche lassen sich zu einer beidseitigen Suche zusammenführen. Dabei werden im Wesentlichen zwei Suchen vom Start- zum Zielknoten und umgekehrt unabhängig voneinander durchgeführt und die Teilergebnisse zur Lösung des SPSP-Problems verwendet.

Wir schreiben abkürzend Iterate_SSSP(s) für den Schleifenrumpf in Algorithmus 3.30 (Schritt 2.1 und Schleife 2.2, d. h. Auswahl des nächsten Knotens und Bearbeitung der von diesem ausgehenden Kanten), also für eine Iteration zur Lösung des SSSP-Problems mit Startknoten s. Analog schreiben wir Iterate_SDSP(z) für eine Iteration zur Lösung des SDSP-Problems mit Zielknoten z. Die Initialisierung Init_SDSP($\cdot$) ist identisch zu der des SSSP-Problems.

Da jetzt je zwei Werte bzw. Mengen D($\cdot$), R und B verwaltet werden müssen, verwenden wir die Indizes s und z zur Kennzeichnung, auf welche der beiden Suchen sich die Prozeduraufrufe jeweils beziehen.

Algorithmus 4.1 *Schema zur Lösung des SPSP durch beidseitige Suche.*

$A_{4.1}.1$ Init_SSSP(s); $R_s := \{s\}$; $B_s := \emptyset$; Init_SDSP(z); $R_z := \{z\}$; $B_z := \emptyset$;
$A_{4.1}.2$ $D_{min} := \infty$;
$A_{4.1}.3$ **while** $\exists$ ein Weg von s nach z, der kürzer als D_{min} sein könnte **do**
$A_{4.1}.3.1$ **do either**
$A_{4.1}.3.1a.1$ Iterate_SSSP(s)
$A_{4.1}.3.1$ **or**
$A_{4.1}.3.1b.1$ Iterate_SDSP(z)
$A_{4.1}.3.1$ **od**;
$A_{4.1}.3.2$ **if** $u \in B_s \cap B_z$ **then** $D_{min} := \min\{D_{min}, D_s(u) + D_z(u)\}$ **fi**
$A_{4.1}.3$ **od**

Wir lassen zunächst offen, wie wir in Schritt $A_{4.1}.3$ entscheiden, ob solch ein Weg noch existieren kann. Statt mit $u := \text{delete\_min}_{s/z}(R_{s/z})$ können wir prinzipiell wieder beliebig Knoten mit $D_{s/z}(\cdot) = d_{s/z}(\cdot)$ wählen. Die Auswahl, ob B_s oder B_z erweitert wird, ist beliebig, sie hat keinen Einfluss auf die Korrektheit des Algorithmus.

Satz 4.2 *Der Algorithmus 4.1 terminiert stets mit* $D_{min} = \text{dist}(s, z)$. *Ist* $T(n, m)$ *der Aufwand des Algorithmus 3.30, so beträgt der Aufwand des Algorithmus 4.1* $O(T(n, m))$ *zzgl. dem Aufwand der Abfrage in* $A_{4.1}.3$.

Beweis: Nach Satz 3.31 werden die Entfernungen $D_s(B_s)$ und $D_z(B_z)$ korrekt berechnet. Die obere Schranke D_{min} wird für jeden neu gefundenen Weg von s nach z in Schritt $A_{4.1}.3.2$ ggf. nach unten angepasst (Aufwand je $O(1)$). Die Terminierung ist gewährleistet, da spätestens abgebrochen werden kann, wenn $z \in B_s$ oder $s \in B_z$ gilt. Der Aufwand beträgt also maximal $2 \cdot T(n, m) + O(n) = O(T(n, m))$ zzgl. dem Aufwand für die Abbruchbedingung. □

Die Abbruchbedingung in Schritt $A_{4.1}.3$ muss in Abhängigkeit von der Auswahl der Knoten u mit $D_{s/z}(u) = d_{s/z}(u)$ gewählt werden. Wir betrachten dazu folgendes Lemma ([Nic66],[Poh71]).

Lemma 4.3 (Dijkstra bidirektional) *Werden in Algorithmus 4.1 die Knoten gemäß Lemma 3.34 ausgewählt, so gelten die beiden Implikationen*

$$B_s \cap B_z \neq \emptyset \;\Rightarrow\; \text{dist}(s, z) = \min_{u \in B_s \cap N(B_z)} \{D_s(u) + D_z(u)\}$$

und

$$B_s \cap B_z \neq \emptyset \;\Rightarrow\; \text{dist}(s, z) = \min_{u \in N(B_s) \cap B_z} \{D_s(u) + D_z(u)\}$$

Beweis: Sei $u \in B_s \cap B_z$. Existiert ein kürzester Weg $w = s \cdots u \cdots z$, so gilt

$$\min_{u' \in B_s \cap N(B_z)} \{\mathrm{D}_s(u') + \mathrm{D}_z(u')\} \leq \mathrm{D}_s(u) + \mathrm{D}_z(u) = \mathrm{dist}(s, z)$$

Existiert ein kürzester Weg $w' = s \cdots u'u'' \cdots z$ mit $u' \in B_s$ und $u'' \in B_z$, so gilt aufgrund der Invariante 3.29 $\mathrm{D}_z(u') \leq \mathrm{d}_z(u'') + \gamma(u''u') = \mathrm{d}_z(u')$ und somit $\mathrm{D}_s(u') + \mathrm{D}_z(u') = \mathrm{dist}(s, z)$. Bleibt der Fall zu untersuchen, dass in jedem kürzesten Weg w'' ein Knoten $v \notin B_s \cup B_z$ existiert. Nach Korollar 3.36 gilt $\mathrm{d}_s(v) \geq \mathrm{d}_s(u)$ und $\mathrm{d}_z(v) \geq \mathrm{d}_z(u)$, somit gilt aber für solch einen Weg w'', dass $\gamma(w'') = \mathrm{d}_s(v) + \mathrm{d}_z(v) \geq \mathrm{d}_s(u) + \mathrm{d}_z(u)$, was im Widerspruch zur Annahme $\mathrm{d}_s(u) + \mathrm{d}_z(u) > \mathrm{dist}(s, z)$ steht.[1]

Die zweite Behauptung folgt durch Vertauschung der Rollen von s und z. □

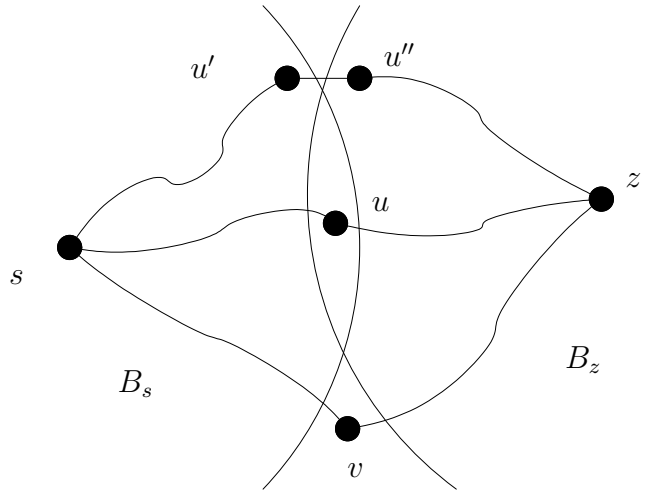


Abbildung 4.1: Illustration des Beweises zu Lemma 4.3

D. h., wir können den Algorithmus 4.1 abbrechen, sobald der erste Knoten in $B_s \cap B_z$ gefunden wurde, und dann in $O(n)$ die kürzeste Entfernung mit Hilfe des Lemmas 4.3 bestimmen.

Die Suche mittels Dijkstra-Algorithmus (Lemma 3.34) sucht ohne zu wissen, in welcher Richtung der Zielknoten liegt. Entscheidet man sich in $\mathrm{A}_{4.1}$.3.1

[1] Die Wege über Knoten $v \notin B_s \cup B_z$ können, obwohl sie keine Verbesserung bringen können, dennoch kürzeste Wege sein, wie man sich leicht an Hand von Gittergraphen klar machen kann. Sucht man beidseitig von $(0, 0)$ nach (k, k), so sind die Knoten auf der Diagonalen $(0, k)$ nach $(k, 0)$ jeweils k von $(0, 0)$ und (k, k) entfernt. Sucht man äquidistant vom Start- und Zielknoten aus, so bricht der Algorithmus ab, wenn einer dieser k Knoten der Diagonalen in $B_s \cap B_z$ liegt, sodass viele weitere kürzeste Wege über Knoten $v \notin B_s \cup B_z$ existieren können. Nur wenn Kantengewichte 0 erlaubt sind, können auch mehrere Knoten $v \notin B_s \cup B_z$ auf einem kürzesten Weg liegen.

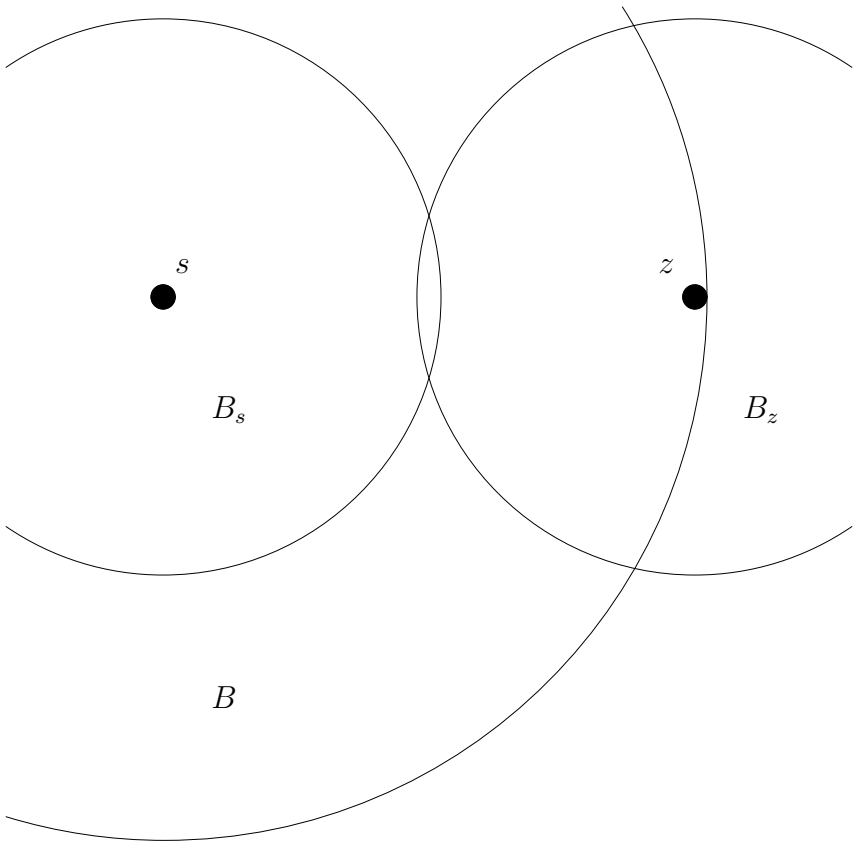


Abbildung 4.2: Verbesserung durch beidseitige Suche?

für den Zweig a oder b abhängig von $\min\{\min(\mathrm{D}_s(R_s)), \min(\mathrm{D}_z(R_z))\}$, so suggeriert die Abbildung 4.2, dass man dabei nur ca. die Hälfte der Knoten betrachten muss, als wenn man die Suche unidirektional wie in Algorithmus 3.30 durchführt.

Betrachtet man Gittergraphen, so werden gegenüber dem unidirektionalen Dijkstra-Algorithmus tatsächlich ziemlich genau die Hälfte der Knoten betrachtet. In mehrdimensionalen Gittergraphen sinkt der Anteil mit jeder weiteren Dimension um den Faktor 2. Es gibt jedoch auch Beispiele, bei denen die bidirektionale Variante mehr Knoten besucht: Dies ist dann der Fall, wenn bei der beidseitigen Variante jenseits des Zielknotens durch die Rückwärtssuche zusätzlich Knoten betrachtet werden und bei der Vorwärtssuche nur wenig Knoten nicht besucht werden [Buc00].

[Poh71] schlägt vor, die Auswahl basierend auf der Anzahl der Knoten in R_s bzw. R_z zu treffen, um so die Anzahl der betrachteten Knoten gering(er) zu halten.

[LR89] zeigen für eine leicht abgewandelte Variante bei Klassen von Zufallsgraphen mit gewissen Anforderungen an die Verteilungsfunktion der Kantengewichte einen Average-Case von $\Theta(\sqrt{n}\log n)$, falls der Graph als Adjazenzliste vorliegt, bei der für jeden Knoten die ausgehenden Kanten der Länge nach sortiert sind. Für die unidirektionale Variante zeigen sie für dieselbe Graphenklasse einen Average-Case von $\Theta(n \log n)$. Liegen die Kantengewichte nicht sortiert vor, so betragen die erwarteten Laufzeiten $\Theta(\frac{m}{n}\sqrt{n}) = \Theta(m/\sqrt{n})$ bzw. $\Theta(\frac{m}{n}n) = \Theta(m)$. In allen Fällen wird dabei $\frac{m}{n} \in \Omega(\log n)$ vorausgesetzt, sodass diese Abschätzungen bei Straßengraphen und anderen Graphen mit $m \in O(n)$ nicht gelten.

4.2 Beidseitige heuristische Suche

Im unidirektionalen Fall werden durch die Verwendung der A^*-Heuristik (Lemma 3.40) stets weniger Knoten betrachtet als bei der Dijkstra-Variante ([Gel77], [Pea84] Seite 85, Korollar aus dem dortigen Korollar zu Theorem 12[2]). Nach Untersuchungen in [Pea84] (Kapitel 5.2, Seiten 140–146) werden z. B. in Gittergraphen bei der Suche nach kürzesten Entfernungen mit dem euklidischen Abstand als Schätzfunktion $\widehat{\mathrm{ed}}(\cdot)$ nur etwa 18% der Knoten betrachtet.

Dies lässt hoffen, dass die bidirektionale Variante des A^*-Algorithmus noch weniger Knoten betrachten muss. Eine Abbruchbedingung wie in Lemma 4.3 gilt aber leider nicht, da wir keine Aussagen über die Entfernungen $\mathrm{d}(V-B)$ und $\mathrm{d}(B)$ wie im Korollar 3.36 machen können. Wir kommen im Abschnitt 4.3 auf die Frage zurück, welche Güte garantiert werden kann, wenn man bei der bidirektionalen heuristischen Suche abbricht, sobald der erste Knoten $v \in B_s \cup B_z$ gefunden wurde.

Lemma 4.4 (A^* bidirektional, [Poh71]) *Werden in Algorithmus 4.1 die Knoten gemäß Lemma 3.40 ausgewählt, so gilt*

$$\max\{\min_{v\in V-B_s}\{\mathrm{D}_s(v)+\widehat{\mathrm{ed}}_z(v)\}, \min_{v\in V-B_z}\{\mathrm{D}_z(v)+\widehat{\mathrm{ed}}_s(v)\}\} \geq \mathrm{D}_{\min} \Rightarrow \mathrm{D}_{\min} = \mathrm{dist}(s,z)$$

[2]Dabei wird davon ausgegangen, dass bei Gleichheit der $\mathrm{D}(\cdot)$ Werte der Zielknoten bevorzugt genommen wird, ansonsten könnte sich bei entsprechenden Auswahlkriterien – und nur bei sehr schlechten Schätzfunktionen – ein Vorteil von Dijkstra gegenüber A^* zeigen.

Beweis: Wir zeigen die Umkehrung dieser Aussage: Angenommen es sei $\mathrm{D}_{\min} > \mathrm{dist}(s,z)$, dann existiert in jedem kürzesten Weg $w = s \cdots v \cdots z$ ein $v \notin B_s \cup B_z$ – sonst wäre nach Lemma 3.1 $\mathrm{D}_{\min} = \mathrm{dist}(s,z)$. Sei v_s der erste Knoten in w mit $v_s \in V - B_s$ und v_z der letzte Knoten in w mit $v_z \in V - B_z$. Dann gilt

$$\begin{array}{rcll} \mathrm{D}_s(v_s) + \widehat{\mathrm{ed}}_z(v_s) & = & \mathrm{d}_s(v_s) + \widehat{\mathrm{ed}}_z(v_s) & \text{Invariante 3.29} \\ & \leq & \mathrm{dist}(s,z) & \widehat{\mathrm{ed}}_z(\cdot) \text{ ist zulässig + Lemma 3.1} \end{array}$$

analog gilt $\mathrm{D}_z(v_z) + \widehat{\mathrm{ed}}_s(v_z) = \mathrm{d}_z(v_z) + \widehat{\mathrm{ed}}_s(v_z) \leq \mathrm{dist}(s,z)$, somit gilt

$$\min_{v \in V - B_s} \{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\} \leq \mathrm{dist}(s,z) \text{ und } \min_{v \in V - B_z} \{\mathrm{D}_z(v) + \widehat{\mathrm{ed}}_s(v)\} \leq \mathrm{dist}(s,z)$$

und es folgt

$$\max\{\min_{v \in V - B_s} \{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\}, \min_{v \in V - B_z} \{\mathrm{D}_z(v) + \widehat{\mathrm{ed}}_s(v)\}\} \leq \mathrm{dist}(s,z) < \mathrm{D}_{\min}$$

mit obiger Annahme $\mathrm{D}_{\min} > \mathrm{dist}(s,z)$. □

D. h., solange auf keinem kürzesten Weg von s nach z ein Knoten v sowohl in B_s als auch in B_z aufgenommen wurde, darf die beidseitige heuristische Suche nicht abgebrochen werden. Leider liegt der erste Knoten, der in beide Baummengen aufgenommen wird, nicht unbedingt auf einem kürzesten Weg, sodass das Abbruchkriterium erst später greift. Selbst wenn ein Knoten gefunden wurde, der $\mathrm{D}_{\min} := \mathrm{dist}(s,z)$ setzt, so kann der Algorithmus dies zu diesem Zeitpunkt noch nicht erkennen, sondern rechnet solange weiter, bis entweder für die Vorwärts- oder für die Rückwärtssuche kein Kandidat mehr existiert, der $\mathrm{D}_{\min}$ verringern könnte.

Folgendes Beispiel lässt erste Zweifel an der Hoffnung aufkommen, dass die beidseitige heuristische Suche der normalen beidseitigen Suche überlegen ist. Im Gegensatz zur unidirektionalen Variante kann man leicht Beispiele konstruieren, bei der der beidseitige A^*-Algorithmus beliebig mehr Knoten in die beiden Baummengen aufnimmt als die beidseitige Dijkstra-Variante (Abbildung 4.3, $\gamma(uv) = \widehat{\mathrm{ed}}(u,v) = \mathrm{d}^{(e)}(u,v)$, $uv \in E$ bzw. $u, v \in V$).

Bevor der (einzige) kürzeste Weg von s nach z mittels beidseitigem A^*-Algorithmus gefunden wird, werden zunächst alle Knoten im oberen Zweig in B_s und alle im unteren Zweig in B_z aufgenommen, während Dikstra beidseitig nach s' den Knoten z' in B_s aufnimmt (sofern $\gamma(s'z') < \gamma(s's'')$) und endet.

Wir bringen die obige Hoffnung nun endgültig zu Fall, indem wir zeigen, dass die beidseitige Anwendung des A^*-Algorithmus gegenüber der unidirektionalen Variante keinen Vorteil bieten kann. Es wird zwar sehr schnell ein Knoten

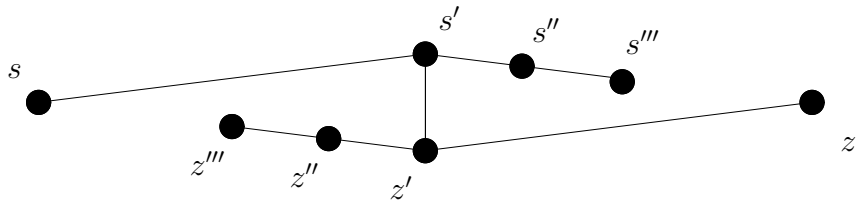


Abbildung 4.3: A^* erzeugt beidseitig u. U. größere Baummengen $B_{s/z}$

eines kürzesten Weges in $B_s \cap B_z$ aufgenommen, der Algorithmus muss dann aber noch lange weiterrechnen, um nachzuweisen, dass andere Wege, die nach der gegebenen Schätzfunktion noch kürzer sein könnten, nicht mehr zu einem besseren Ergebnis führen. Dieses Problem wurde erstmals von [KK97] richtig erkannt, dort werden aber nur triviale Abschätzungen bzgl. der Anzahl der betrachteten Knoten bei der bidirektionalen gegenüber der unidirektionalen heuristischen Suche angegeben. Lange Zeit wurde als Ursache der unerwartet langen Laufzeiten angenommen, dass sich die Mengen B_s und B_z nicht in der Mitte treffen, sondern aneinander vorbeilaufen (z. B. [Poh71]).[3] De facto durchdringen sie sich aber und es wird ein großer Teil des Graphen doppelt untersucht. Die Verbesserungsvorschläge, die diesbzgl. gemacht wurden (z. B. [PP84] und [dCS77], [dC83]), lassen den Algorithmus zwar frühzeitig abbrechen, können jedoch nicht mehr garantieren, den wirklich kürzesten Weg zu finden.

Wir zeigen zunächst folgendes Lemma, das eine wichtige Aussage zur Reihenfolge bringt, in der die Knoten in die Mengen B_s und B_z aufgenommen werden.

Lemma 4.5 (Kommutativität) *Die Folge $f = f_1, f_2, \ldots, f_k \in \{b_s, b_z\}^k$ gebe an, in welcher Reihenfolge die Mengen B_s und B_z erweitert werden. Nach dem entsprechenden Ausführen gelte die Abbruchbedingung aus Lemma 4.4:*

$$\max\{\min_{v \in V - B_s}\{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\}, \min_{v \in V - B_z}\{\mathrm{D}_z(v) + \widehat{\mathrm{ed}}_s(v)\}\} \geq \mathrm{D}_{\min}$$

Dann gilt, dass die Bedingung nach der Ausführung einer beliebigen Permutation der Folge f erfüllt ist, insbesondere auch nach der Folge $f' = b_s^{|f|_{b_s}} b_z^{|f|_{b_z}}$.

[3] Zu erklären ist dies höchstens damit, dass die beidseitige heuristische Suche hauptsächlich auf Graphen betrachtet wurde, die erst während der Wegesuche konstruiert wurden (z. B. die Lösung von Verschiebepuzzeln, weitere ähnliche Anwendungen in [Pea84]).

Beweis: Die Baummenge B_z hat keinen Einfluss auf den Wert $\min_{v \in V - B_s}\{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\}$, d. h., führt man alle Schritte für die Menge B_s zuerst aus, so nimmt das Minimum denselben Wert an. Analog gilt dies für $\min_{v \in V - B_z}\{\mathrm{D}_z(v) + \widehat{\mathrm{ed}}_s(v)\}$. Da $\mathrm{D}_{s/z}(v) = \mathrm{d}_{s/z}(v)$ für $v \in B_{s/z}$ nimmt auch das $\mathrm{D}_{\min}$ denselben Wert an (es gilt $\mathrm{D}_{\min} = \mathrm{dist}(s, z)$). Die Bedingung ist folglich nach dem Abarbeiten von f' ebenfalls erfüllt. □

Wir benötigen neben dem Lemma 3.42 noch eine weitere Monotonieeigenschaft der A^*-Heuristik.

Lemma 4.6 *Sei $w = v_0 v_1 \cdots v_k$ ein kürzester Weg von $s = v_0$ nach $z = v_k$, so ist $\mathrm{d}_s(v_i) + \widehat{\mathrm{ed}}_z(v_i)$ bzgl. i monoton steigend.*

Beweis: Aufgrund der Konsistenz-Eigenschaft von $\widehat{\mathrm{ed}}_z(\cdot)$ gilt $\mathrm{d}_s(v_i) + \widehat{\mathrm{ed}}_z(v_i) \leq \mathrm{d}_s(v_i) + \gamma(v_i v_{i+1}) + \widehat{\mathrm{ed}}_z(v_{i+1}) = \mathrm{d}_s(v_{i+1}) + \widehat{\mathrm{ed}}_z(v_{i+1})$. □

Wir können damit nun den folgenden Satz beweisen.

Satz 4.7 *Für jeden Ablauf der beidseitigen heuristischen Suche nach Algorithmus 4.1 gilt $|B_s| + |B_z| \geq \min(|B'_s|, |B'_z|)$, wobei die B'_s, B'_z einem Ablauf der einseitigen heuristischen Suche nach Algorithmus 3.30 mit denselben Funktionen $\widehat{\mathrm{ed}}_z(\cdot)$ bzw. $\widehat{\mathrm{ed}}_s(\cdot)$ entsprechen.*

Beweis: Falls die beidseitige Suche durch $z \in B_s$ abgebrochen wurde, so gilt nach Lemma 4.5, dass zunächst alle Schritte für B_s hätten durchgeführt werden können. Damit gilt $|B'_s| = |B_s| \leq |B_s| + |B_z|$ – analog für $s \in B_z$.

Andernfalls muss einerseits wenigstens ein Knoten eines kürzesten Wegs sowohl in B_s als auch in B_z aufgenommen werden, damit das $\mathrm{D}_{\min}$ auf $\mathrm{dist}(s, z)$ gesetzt wird, und in B_s müssen (nach Lemma 3.42) alle Knoten v mit $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) < \mathrm{dist}(s, z)$ aufgenommen worden sein, sodass dann gemäß Lemma 4.4 $\min_{v \in V - B_s}\{\mathrm{D}_s(v) + \widehat{\mathrm{ed}}_z(v)\} \geq \mathrm{dist}(s, z)$ gilt (oder analoges für B_z – wir betrachten hier weiter den Fall für B_s). Seien also zunächst alle Knoten mit $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) < \mathrm{dist}(s, z)$ in B_s aufgenommen worden und sei $w = v_0 v_1 \cdots v_k$ ein kürzester Weg von $s = v_0$ nach $z = v_k$, bei dem die Anzahl der Knoten mit $\mathrm{d}_s(v_j) + \widehat{\mathrm{ed}}_z(v_j) = \mathrm{dist}(s, z)$ minimal ist. Sei v_i der erste Knoten auf diesem Weg mit $\mathrm{d}_s(v_i) + \widehat{\mathrm{ed}}_z(v_i) = \mathrm{dist}(s, z)$ – nach Lemma 4.6 gilt dies auch für die verbleibenden Knoten $v_{i+1}, \ldots, v_k$. Einerseits wird nach Invariante 3.29 das $\min_{v' \in V - (B_s \cup \{v\})}\{\mathrm{D}_s(v') + \widehat{\mathrm{ed}}_z(v')\}$ durch v_i angenommen und es könnten die Knoten $v_i, \ldots, v_k$ nacheinander in B_s aufgenommen werden, womit die Suche wiederum durch $z \in B_s$ abgebrochen würde. Andernfalls muss zumindest ein Knoten $v' \in \{v_i, \ldots, v_{k-1}\}$

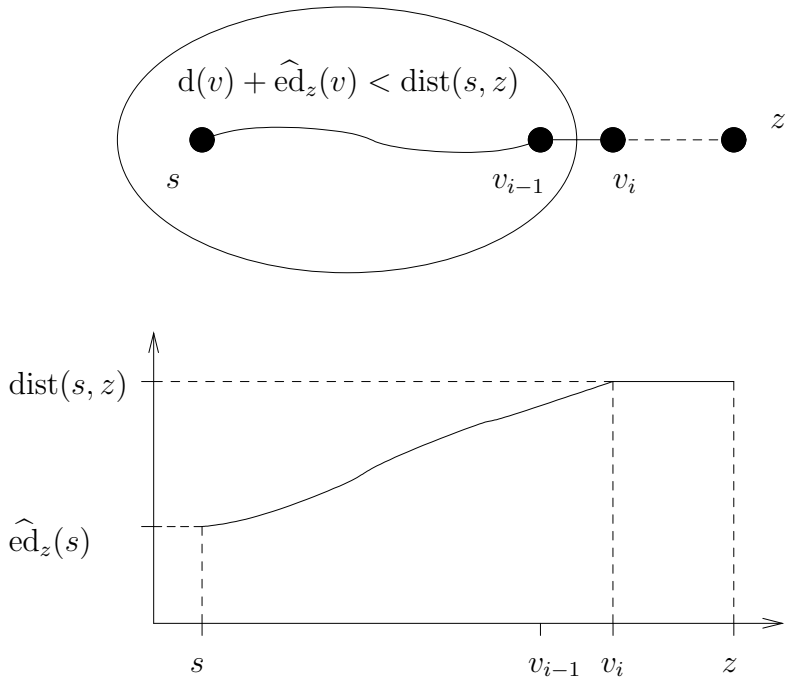


Abbildung 4.4: Illustration des Beweises zu Satz 4.7

sowohl in B_s als auch in B_z aufgenommen werden. Unabhängig davon, in welcher Reihenfolge die Knoten $v_i, \ldots, v_{k-1}$ in B_s oder B_z aufgenommen werden, gilt $|B_s| + |B_z| \geq |\{v|\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) < \mathrm{dist}(s,z)\}| + k - i + 2$, während ein Ablauf einer einseitigen heuristischen Suche existiert, die mit $|B'_s| = |\{v|\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) < \mathrm{dist}(s,z)\}| + k - i + 1$ endet. □

Wir sehen, dass eine beidseitige heuristische Suche nur dann Vorteile bringen kann, wenn für viele Knoten $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) = \mathrm{dist}(s,z)$ gilt und die Auswahl der Knoten bei obiger Gleichheit zu Gunsten der beidseitigen Suche ausfällt. Gilt nur für den Zielknoten $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) = \mathrm{dist}(s,z)$ (und analoges für den Startknoten), wie es z. B. bei Straßengraphen mit Luftlinienabstand als Schätzfunktion meistens der Fall sein wird, so ist keine Verbesserung durch die beidseitige Variante gegenüber der besseren der einseitigen Varianten – Suche Start zum Ziel oder Suche Ziel zum Start – möglich. Ist aufgrund der Problemstruktur denkbar, dass sich der Aufwand der Suche Start zum Ziel oder Suche Ziel zum Start stark unterscheidet, so ist ein paralleles und unabhängiges Ausführen beider Suchen der beidseitigen heuristischen Suche aufgrund des geringeren Overheads (einfaches Abbruchkriterium, einfachere Verwaltung der Knotenmengen) vorzuziehen.

Der Hauptvorteil des frühen Findens des kürzesten Weges wird dadurch verschenkt, dass große Teile des Graphen doppelt betrachtet werden. Unabsichtlich hat die Verbesserung nach [Kwa89] dieses Problem folgendermaßen gelindert. Sobald ein Knoten $v \in B_z$ auch in B_s aufgenommen wurde, brauchen deren Nachfolger, die ebenfalls schon in B_z sind, nicht betrachtet zu werden, da die kürzeste Entfernung $\mathrm{d}_z(v)$ gemäß Lemma 3.40 korrekt berechnet wurde und somit die Summe $\mathrm{d}_s(v) + \mathrm{dist}(v,u) + \mathrm{d}_z(u)$ für alle Knoten $u \in B_z$ nicht kleiner als $\mathrm{d}_s(v) + \mathrm{d}_z(v)$ werden kann (analog bei der Suche von z in Richtung s). Auch für diese Variante (sie nennt sich BS^*) gilt jedoch, dass eine der beiden unidirektionalen A^* Suchen schneller sein muss als die beidseitige BS^* Suche. Analog zum Beweis zu Satz 4.7 müssen (bzgl. Suchrichtung s nach z) alle Knoten v mit $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) < \mathrm{dist}(s,z)$ in B_s (oder analog in B_z) aufgenommen worden sein, sodass auch hier die gleiche Aussage wie in Satz 4.7 gilt.

Wird, wie in [Kwa89] vorgeschlagen, das $\mathrm{D}_{\min}$ auch aktualisiert, wenn ein Knoten durch ein Update in $R_s \cap R_z$ aufgenommen wurde, so können gegenüber dem obigen Beweis maximal je ein Knoten weniger in B_s und B_z aufgenommen werden, bevor die Abbruchbedingung erfüllt ist. Somit kann gegenüber der unidirektionalen Variante ein Gesamtvorteil von maximal einem Knoten entstehen. Der Mehraufwand, der durch ein vielfach häufigeres Aktualisieren des $\mathrm{D}_{\min}$ entsteht ($O(m)$ zusätzliche Tests nach decrease_key$(\cdot)$-Aufrufen), lohnt sich demzufolge nicht.

4.3 Approximationen durch beidseitige heuristische Suche

Bei der beidseitigen heuristischen Suche wird im Allgemeinen ein großer Teil des Graphen doppelt bearbeitet, um die meist schon früh gefundene Lösung als optimal beweisen zu können.

Wir betrachten nun, welche Güte Näherungslösungen haben, wenn die beidseitige heuristische Suche abgebrochen wird, sobald der erste Knoten in beide Baummengen aufgenommen wurde. Ein Abbruch, sobald ein Knoten in beiden Randmengen aufgenommen wurde, führt offensichtlich zu beliebig schlechten Resultaten.

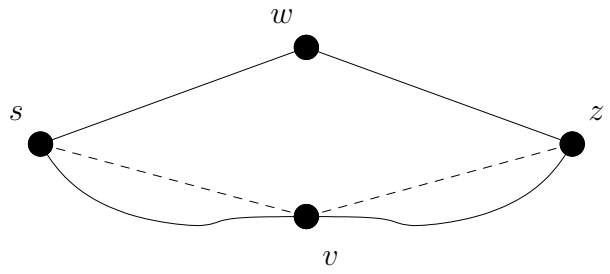
Satz 4.8 (Approximation kürzester Wege durch bidirekt. Suche)
Werden in Algorithmus 4.1 die Knoten gemäß Lemma 3.40 mit konsistenten Schätzfunktionen $\widehat{\mathrm{ed}}_{s/z}(\cdot) \leq \mathrm{d}_{s/z}(\cdot) \leq (1+\varepsilon) \cdot \widehat{\mathrm{ed}}_{s/z}(\cdot)$ *aufgenommen, so gilt*

$$v \in B_s \cap B_z \;\Rightarrow\; \mathrm{d}_s(v) + \mathrm{d}_z(v) \leq \left(1 + \frac{\varepsilon}{2+\varepsilon}\right) \cdot \mathrm{dist}(s,z)$$

Beweis: Mit $v \in B_s$ gilt nach Lemma 3.42, dass $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}_z(v) \leq \mathrm{dist}(s,z)$ und somit $\mathrm{d}_s(v) + \frac{1}{1+\varepsilon} \cdot \mathrm{d}_z(v) \leq \mathrm{dist}(s,z)$, analog gilt mit $v \in B_z$, dass $\frac{1}{1+\varepsilon} \cdot \mathrm{d}_s(v) + \mathrm{d}_z(v) \leq \mathrm{dist}(s,z)$. Somit folgt

$$(\mathrm{d}_s(v) + \mathrm{d}_z(v)) \cdot \left(1 + \frac{1}{1+\varepsilon}\right) = (\mathrm{d}_s(v) + \mathrm{d}_z(v)) \cdot \left(\frac{2+\varepsilon}{1+\varepsilon}\right) \leq 2 \cdot \mathrm{dist}(s,z)$$

und damit direkt die Behauptung. □



$$\gamma(s,w) = \gamma(w,z) = \widehat{\mathrm{ed}}_s(w) = \widehat{\mathrm{ed}}_z(w) = \tfrac{1}{2}$$
$$\gamma(s,v) = \gamma(v,z) = \tfrac{1+\varepsilon}{2+\varepsilon}$$
$$\widehat{\mathrm{ed}}_s(v) = \widehat{\mathrm{ed}}_z(v) = \tfrac{1}{2+\varepsilon}$$

Abbildung 4.5: Worst-Case Beispiel zu Satz 4.8

Die Voraussetzung der Konsistenz ist notwendig, da sonst für die ausgewählten Knoten $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ und damit auch die Güte der Approximation nicht garantiert werden könnte.

Der Worst-Case lässt sich relativ leicht erreichen: Der kürzeste Weg von s nach z in Abbildung 4.5 führt über den Knoten w und hat die Länge $\gamma(swz) = 1$. Die geschätzte Entfernung über den Knoten v beträgt $\gamma(sv)+\widehat{\mathrm{ed}}_z(v) = \gamma(zv)+\widehat{\mathrm{ed}}_s(v)$, ebenfalls 1, sodass v vor w in B_s und B_z aufgenommen werden kann. Der approximierte Weg hat die Länge $\gamma(svz) = \frac{2+2\varepsilon}{2+\varepsilon} = 1 + \frac{\varepsilon}{2+\varepsilon}$.

Bei der Anwendung in Straßengraphen kann der Luftlinienabstand als Schätzfunktion verwendet werden, ferner entspricht die Entfernung zwischen zwei Kreuzungen meist (annähernd) der Luftlinie. Ggf. kann man dies durch Einfügen von Knoten mit Grad 2 erreichen. Man vermutet zunächst, dass für solche eingeschränkten Graphen bessere Abschätzungen möglich sein müssten. Entgegen der Intuition gibt es aber selbst für solche euklidischen Graphen Beispiele, bei denen der Worst-Case aus Satz 4.8 angenommen wird (Abbildung 4.6).

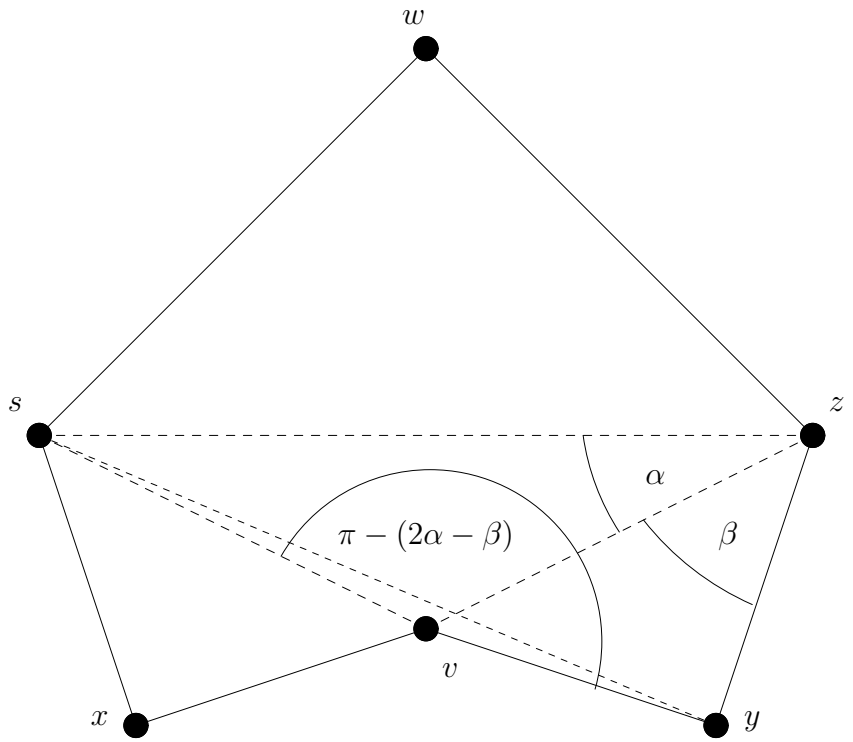


Abbildung 4.6: Euklidischer Graph als Worst-Case Beispiel zu Satz 4.8

Mit einem Parameter $\varepsilon > 0$ seien die Kantenlängen und Schätzentfernungen wie folgt gewählt:

$$\gamma(s,w) = \gamma(w,z) = \frac{1+\varepsilon}{2}$$

$$\gamma(s,x) = \gamma(x,v) = \gamma(v,y) = \gamma(y,z) = \frac{(1+\varepsilon)^2}{2(2+\varepsilon)}$$

$$\widehat{\mathrm{ed}}(s,v) = \widehat{\mathrm{ed}}(v,z) = \frac{1+\varepsilon}{2+\varepsilon}$$

Wir schreiben $\widehat{\mathrm{ed}}(u,v)$ für $\widehat{\mathrm{ed}}_u(v) = \widehat{\mathrm{ed}}_v(u) = \mathrm{d}^{(e)}(u,v)$. Der kürzeste Weg von s nach z führt über w und hat die Länge $\mathrm{dist}(s,z) = (1+\varepsilon)$ (ebenso ist $\mathrm{d}_s(w) + \widehat{\mathrm{ed}}(w,z) = \mathrm{d}_z(w) + \widehat{\mathrm{ed}}(w,s) = \mathrm{dist}(s,z)$). Es gilt weiter

$$\begin{aligned}
\mathrm{d}_s(x) + \widehat{\mathrm{ed}}(x,z) &\leq \mathrm{d}_s(x) + \gamma(x,v) + \widehat{\mathrm{ed}}(v,z) \\
&= \mathrm{d}_s(v) + \widehat{\mathrm{ed}}(v,z) \\
&= \left(2\frac{(1+\varepsilon)^2}{2(2+\varepsilon)} + \frac{1+\varepsilon}{2+\varepsilon}\right) \\
&= (1+\varepsilon)\cdot\left(\frac{1+\varepsilon}{2+\varepsilon} + \frac{1}{2+\varepsilon}\right) \\
&= \mathrm{dist}(s,z)
\end{aligned}$$

Es können also die Knoten x und v vor w in B_s aufgenommen werden, analog können y und v vor w in B_z aufgenommen werden. Die Approximation ermittelt eine Weglänge von

$$4\cdot\frac{(1+\varepsilon)^2}{2(2+\varepsilon)} = \frac{2(1+\varepsilon)^2}{2+\varepsilon}\cdot\left(\frac{1}{1+\varepsilon}\cdot\mathrm{dist}(s,z)\right) = \left(1+\frac{\varepsilon}{2+\varepsilon}\right)\cdot\mathrm{dist}(s,z)$$

was genau dem Worst-Case von Satz 4.8 entspricht.

Es bleibt für das Beispiel zu zeigen, dass auch für alle relevanten Knotenpaare die Voraussetzung des Satzes 4.8 erfüllt ist. Es gilt

$$\mathrm{dist}(s,v) = 2\cdot\frac{(1+\varepsilon)^2}{2(2+\varepsilon)} = \frac{(1+\varepsilon)^2}{2+\varepsilon}\cdot\left(\frac{2+\varepsilon}{1+\varepsilon}\cdot\widehat{\mathrm{ed}}(s,v)\right) = (1+\varepsilon)\cdot\widehat{\mathrm{ed}}(s,v)$$

analog $\mathrm{dist}(v,z) = (1+\varepsilon)\cdot\widehat{\mathrm{ed}}(v,z)$. Für das Paar (s,y) (und analog für (x,z)) gilt mit dem Kosinussatz und $\cos(\pi-(2\alpha-\beta)) = -\cos(2\alpha-\beta)$

$$\widehat{\mathrm{ed}}(s,y)^2 = \widehat{\mathrm{ed}}(s,v)^2 + \widehat{\mathrm{ed}}(v,y)^2 + 2\cdot\widehat{\mathrm{ed}}(s,v)\cdot\widehat{\mathrm{ed}}(v,y)\cdot\cos(2\alpha-\beta)$$

Für die Winkel α und β sind

$$\cos\alpha = \frac{1+\frac{1}{2}\varepsilon}{1+\varepsilon} = \frac{1}{1+\frac{\varepsilon}{2+\varepsilon}} > \frac{1}{2}$$

$$\cos\beta = \frac{1}{1+\varepsilon}$$

Es gilt mit $1 \geq \cos\alpha \geq \cos\beta > 0$ und

$$\begin{aligned}
\cos(2\alpha) = 2(\cos\alpha)^2 - 1 &= \frac{2\cdot(2+\varepsilon)^2 - 4(1+\varepsilon)^2}{4(1+\varepsilon)^2} \\
&= \frac{2-\varepsilon^2}{2(1+\varepsilon)^2} \leq \frac{1}{(1+\varepsilon)^2} \leq \frac{1}{1+\varepsilon} = \cos\beta
\end{aligned}$$

dass $0 \leq \alpha \leq \beta \leq 2\alpha$ (also auch $0 \leq 2\alpha - \beta \leq \beta$) und somit

$$\cos(2\alpha - \beta) \geq \cos\beta = \frac{1}{1+\varepsilon}$$

damit also

$$\begin{aligned}
\widehat{\mathrm{ed}}(s,y)^2 &\geq \left(\frac{1+\varepsilon}{2+\varepsilon}\right)^2 + \left(\frac{(1+\varepsilon)^2}{2(2+\varepsilon)}\right)^2 + 2 \cdot \frac{(1+\varepsilon)}{(2+\varepsilon)} \cdot \frac{(1+\varepsilon)^2}{2(2+\varepsilon)} \cdot \frac{1}{1+\varepsilon} \\
&= \frac{4(1+\varepsilon)^2 + (1+\varepsilon)^4 + 4(1+\varepsilon)^2}{4(2+\varepsilon)^2} \\
&\geq \frac{4(1+\varepsilon)^2 + (1+\varepsilon)^2 + 4(1+\varepsilon)^2}{4(2+\varepsilon)^2} \\
&= \frac{9(1+\varepsilon)^4}{4(2+\varepsilon)^2} \cdot \frac{1}{(1+\varepsilon)^2} \\
&= \left(3 \cdot \frac{(1+\varepsilon)^2}{2(2+\varepsilon)}\right)^2 \cdot \frac{1}{(1+\varepsilon)^2} \\
&= \frac{\mathrm{dist}(s,y)^2}{(1+\varepsilon)^2}
\end{aligned}$$

und somit ist die Bedingung $(1+\varepsilon) \cdot \widehat{\mathrm{ed}}(s,y) \geq \mathrm{dist}(s,y)$ erfüllt.[4]

Interessant sind solche Approximationen z. B. bei Straßengraphen. Dort kann man aber nicht immer garantieren, dass sich die tatsächliche Entfernung und die Schätzfunktion um höchstens einen kleinen Faktor $(1+\varepsilon)$ unterscheiden (z. B. bei Städten links und rechts eines Flusses, die nicht direkt durch eine Brücke verbunden sind). Auch wenn sich die Schätzfunktion für manche Knotenpaare beliebig stark von der tatsächlichen Entfernung unterscheidet, lässt sich mit der beidseitigen Suche eine gute Approximation erreichen.

Satz 4.9 (Approximation kürzester Wege durch bidirek. Suche II) *Werden in Algorithmus 4.1 die Knoten gemäß Lemma 3.40 aufgenommen, so gilt in euklidischen Graphen mit dem euklidischen Abstand als (konsistenter) Schätzfunktion* $\widehat{\mathrm{ed}}(\cdot)$ *für das Knotenpaar* (s,z) *mit* $\mathrm{dist}(s,z) = (1+\varepsilon) \cdot \widehat{\mathrm{ed}}(s,z)$*, dass*

$$v \in B_s \cap B_z \;\Rightarrow\; \mathrm{d}_s(v) + \mathrm{d}_z(v) \leq \left(1 + \frac{\varepsilon}{1+\varepsilon}\right) \cdot \mathrm{dist}(s,z)$$

[4] Die Abschätzung gilt auch für das Knotenpaar (x,y), da $\mathrm{dist}(x,y) = \frac{1}{\cos(\beta-\alpha)} \cdot \widehat{\mathrm{ed}}(x,y) \leq \frac{1}{\cos\beta} \cdot \widehat{\mathrm{ed}}(x,y) = (1+\varepsilon) \cdot \widehat{\mathrm{ed}}(x,y)$, da $0 < \cos\beta \leq \cos\alpha \leq 1$ und somit $\beta > \alpha$ für beliebige ε. Um die Abschätzung auch noch für die verbleibenden Knotenpaare zu erfüllen, kann man Kanten zwischen w und den drei Knoten x, v und y einfügen. Diese haben keinen Einfluss auf den kürzesten Weg von s nach z.

Beweis: Mit $v \in B_s$ gilt nach Lemma 3.42, dass $\mathrm{d}_s(v) + \widehat{\mathrm{ed}}(v,z) \leq \mathrm{dist}(s,z)$, analog gilt mit $v \in B_z$, dass $\mathrm{d}_z(v) + \widehat{\mathrm{ed}}(v,s) \leq \mathrm{dist}(s,z)$. Mit der Dreiecksungleichung für euklidische Entfernungen folgt

$$\mathrm{d}_s(v) + \mathrm{d}_z(v) + \widehat{\mathrm{ed}}(s,z) \leq 2 \cdot \mathrm{dist}(s,z)$$

und mit $\mathrm{dist}(s,z) = (1+\varepsilon) \cdot \widehat{\mathrm{ed}}(s,z)$ die Behauptung. □

Verwendet man z. B. in Straßengraphen den Luftlinienabstand als Schätzfunktion, so gilt in weiten Teilen, dass der tatsächliche Weg höchstens 20%, also $\frac{1}{5}$ länger als die Luftlinie ist. Mit Satz 4.9 erhält man eine Approximation, die maximal $\frac{0.2}{1.2} = \frac{1}{6}$ über der tatsächlichen Entfernung liegt. Gilt die 20% Abschätzung für alle Knotenpaare, bei denen ein Knoten der Start- oder Zielknoten ist, kann die Approximation mit Satz 4.8 sogar auf $\frac{0.2}{2.2} = \frac{1}{11}$ genau garantiert werden. Empirisch dürfte im Mittel eine deutlich geringere Abweichung zu erwarten sein.

4.4 Verbesserung der beidseitigen Suche durch Potentialfunktionen

Während die klassische beidseitige Suche mit der A^*-Heuristik durch die starke Durchdringung der berechneten Baummengen und den dadurch großen Anteil doppelter Berechnungen keine Verbesserung bringt, werden wir nun zeigen, wie wir über den Umweg der Potentialfunktionen einen Weg finden, die A^*-Heuristik auch beidseitig sinnvoll anzuwenden.

Zur Erinnerung: Ein Kürzeste-Wege-Problem kann gelöst werden, indem die Kantengewichte des Graphen durch eine beliebige Potentialfunktion π verändert werden, das Problem auf dem reduzierten Graphen G_π gelöst wird und dann die kürzesten Entfernungen über die Potentialfunktion wieder auf den Graphen G übertragen werden. Ebenso haben wir in Abschnitt 3.5 gezeigt, dass der A^*-Algorithmus mit Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ und der Dijkstra-Algorithmus mit Potentialfunktion $\pi(\cdot) := -\widehat{\mathrm{ed}}_z(\cdot)$ äquivalent sind.

Die Idee ist also nun, die Kantengewichte durch eine Potentialfunktion π, die die Schätzfunktionen beider Richtungen $\widehat{\mathrm{ed}}_z(\cdot)$ und $\widehat{\mathrm{ed}}_s(\cdot)$ in angemessener Weise berücksichtigt, zu verändern, um dann mit der beidseitigen Variante nach Dijkstra den kürzesten Weg von s nach z zu berechnen und über die Potentialfunktion das Ergebnis wieder auf den Graphen G zu übertragen. Anschaulich verändert die Potentialfunktion die Kantengewichte so, dass die Entfernung zwischen s und z in G_π abnimmt – im unidirektionalen Fall um

den Wert $\widehat{\text{ed}}_z(s)$ – und so mittels des beidseitigen Dijkstra-Algorithmus der kürzeste Weg schneller gefunden wird als in G.

Die einzige Bedingung, die an die zu verwendene Potentialfunktion π zu stellen ist, ist die Konsistenz. Damit sind die Kantengewichte in G_π positiv und der beidseitige Dijkstra-Algorithmus löst in G_π das SPSP-Problem effizient. Wir geben drei Möglichkeiten für die Potentialfunktion π an.

In Abschnitt 3.5 wurde bereits $\pi(\cdot) = -\widehat{\text{ed}}_z(\cdot)$ behandelt. Aus der Konsistenz von $\widehat{\text{ed}}_z(\cdot)$, d. h. $\widehat{\text{ed}}_z(z) = 0$ und $\widehat{\text{ed}}_z(u) \leq \gamma(uv) + \widehat{\text{ed}}_z(v),\ uv \in E$ folgt direkt die Konsistenz von $\pi(\cdot)$, da

$$\gamma_\pi(uv) = \pi(u) + \gamma(uv) - \pi(v) = -\widehat{\text{ed}}_z(u) + \gamma(uv) + \widehat{\text{ed}}_z(v) \geq 0$$

Obwohl die Schätzfunktion $\widehat{\text{ed}}_s(\cdot)$ unberücksichtigt bleibt, lässt sich so in G_π mit der einfachen beidseitigen Suche das SPSP-Problem lösen, der kürzeste Weg von s nach z im Originalgraphen hat die Länge $\text{dist}_\pi(s,z) + \widehat{\text{ed}}_z(s)$.

Die zweite Möglichkeit ist eine Verallgemeinerung des Ansatzes von Ikeda et. al. ([IHI94]).

Lemma 4.10 *Sind $\widehat{\text{ed}}_z(\cdot)$ und $\widehat{\text{ed}}_s(\cdot)$ konsistent, so ist die Potentialfunktion $\pi(\cdot) = \alpha \cdot \widehat{\text{ed}}_s(\cdot) - (1-\alpha) \cdot \widehat{\text{ed}}_z(\cdot)$ ebenfalls konsistent.*

Beweis: Es gilt $\widehat{\text{ed}}_z(z) = 0$ und $\widehat{\text{ed}}_z(u) \leq \gamma(uv) + \widehat{\text{ed}}_z(v),\ uv \in E$ und $\widehat{\text{ed}}_s(s) = 0$ und $\widehat{\text{ed}}_s(v) \leq \gamma(uv) + \widehat{\text{ed}}_s(u),\ uv \in E$.[5] Es gilt für $0 \leq \alpha \leq 1$

$$\begin{aligned}
\gamma_\pi(uv) &= \pi(u) + \gamma(uv) - \pi(v) \\
&= \gamma(uv) + \alpha \cdot (\widehat{\text{ed}}_s(u) - \widehat{\text{ed}}_s(v)) - (1-\alpha) \cdot (\widehat{\text{ed}}_z(u) - \widehat{\text{ed}}_z(v)) \\
&\geq \gamma(uv) + \alpha \cdot (-\gamma(uv)) - (1-\alpha) \cdot \gamma(uv) \\
&= 0
\end{aligned}$$

Der kürzeste Weg von s nach z im Originalgraphen G hat die Länge

$$\begin{aligned}
\text{dist}(s,z) &= \text{dist}_\pi(s,z) - \alpha \cdot (\widehat{\text{ed}}_s(s) - \widehat{\text{ed}}_s(z)) + (1-\alpha) \cdot (\widehat{\text{ed}}_z(s) - \widehat{\text{ed}}_z(z)) \\
&= \text{dist}_\pi(s,z) + \alpha \cdot \widehat{\text{ed}}_s(z) + (1-\alpha) \cdot \widehat{\text{ed}}_z(s)
\end{aligned}$$

□

[5] Da $\widehat{\text{ed}}_s(\cdot)$ konsistent bzgl. des Graphen ist, in dem die Kanten umgekehrt orientiert sind – in dem also die Kürzeste-Wege-Suche rückwärts (SDSP zu z) durchgeführt wird – sind die Rollen von u und v in der Konsistenzeigenschaft vertauscht.

Ikeda et. al. ([IHI94]) benutzen $\alpha = \frac{1}{2}$, prinzipiell kann α aber beliebig zwischen 0 und 1 gewählt werden. Diese Vorgehensweise interpretieren sie als beidseitige heuristische Suche zwischen Start- und Zielknoten mit Schätzfunktionen $\widehat{\mathrm{ed}}_z'(\cdot) = -\widehat{\mathrm{ed}}_s'(\cdot) = \frac{1}{2}(\widehat{\mathrm{ed}}_z(\cdot) - \widehat{\mathrm{ed}}_s(\cdot))$.

Eine dritte Möglichkeit geben Goldberg und Harrelson ([GH04]) in einem kürzlich veröffentlichten Bericht an: Die Funktion $\pi(\cdot) = \widehat{\mathrm{ed}}_s(\cdot)$ gibt eine untere Schranke für die Entfernung vom Knoten s an und reduziert die Kantengewichte in G_π um z herum bzgl. der beidseitigen Suche sehr effektiv: die Gewichte der Kanten aus der Richtung des Startknotens werden reduziert, die der Kanten aus der entgegengesetzten Richtung werden vergrößert, die Gewichte der tangential zur Gerade Startknoten-Zielknoten liegenden Kanten bleiben ungefähr gleich, sodass bzgl. der beidseitigen Suche hauptsächlich die Kanten, die wahrscheinlich auf dem kürzesten Weg von s nach z liegen, reduziert werden. In der Nähe von s werden alle ausgehenden Kanten gleichermaßen verkürzt, sodass hier eine andere Funktion verwendet werden sollte. [GH04] merken an, dass $\pi(\cdot) = \widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot)$ ebenfalls eine untere Schranke für die Entfernung vom Knoten s angibt, die zwar nicht unbedingt sehr gut ist, jedoch in der Nähe des Knotens s die Kantengewichte in G_π bzgl. beidseitiger Suche zielgerichtet und damit besser reduziert als $\widehat{\mathrm{ed}}_s(\cdot)$. Die Vorteile der beiden Potentialfunktionen versuchen sie mit Hilfe des folgenden Lemmas zu vereinigen.

Lemma 4.11 *Sind die Potentialfunktionen $\pi_1(\cdot)$ und $\pi_2(\cdot)$ konsistent, so ist die Potentialfunktion $\pi(\cdot) = \max\{\pi_1(\cdot), \pi_2(\cdot)\}$ ebenfalls konsistent.*

Beweis: Gilt für $uv \in E$, dass $\pi_1(u) \geq \pi_2(u)$ und $\pi_1(v) \geq \pi_2(v)$, so ist $\pi(\cdot)$ konsistent, da $\pi_1(\cdot)$ konsistent ist, analog, falls $\pi_2(\cdot) \geq \pi_1(\cdot)$. Ist nun $\pi_1(u) > \pi_2(u)$, aber $\pi_1(v) < \pi_2(v)$, so gilt

$$\begin{aligned}\gamma_\pi(uv) &= \pi(u) + \gamma(uv) - \pi(v)\\ &= \pi_1(u) + \gamma(uv) - \pi_2(v) > \pi_2(u) + \gamma(uv) - \pi_2(v) \geq 0\end{aligned}$$

– analog, falls $\pi_1(u) < \pi_2(u)$ und $\pi_1(v) > \pi_2(v)$. □

Die Potentialfunktion $\pi'(\cdot) = \widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot) + c,\ c \in \mathbb{R}$ ist konsistent, wenn die Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ konsistent ist, da dann

$$\begin{aligned}\gamma_{\pi'}(uv) &= \pi'(u) + \gamma(uv) - \pi'(v)\\ &= \widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(u) + c + \gamma(uv) - \widehat{\mathrm{ed}}_z(s) + \widehat{\mathrm{ed}}_z(v) - c\\ &= -\widehat{\mathrm{ed}}_z(u) + \gamma(uv) + \widehat{\mathrm{ed}}_z(v)\\ &\geq 0\end{aligned}$$

Somit ist die von [GH04] verwendete Potentialfunktion

$$\pi(\cdot) = \max\{\widehat{\mathrm{ed}}_s(\cdot), \widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot) + c\},\ c \in \mathbb{R}$$

nach Lemma 4.11 ebenfalls konsistent. [GH04] argumentieren nun, dass in der Nähe von s der zweite Term dominiert, in der Nähe von z der erste, sodass dort jeweils die Kantengewichte zielgerichtet reduziert werden sollen. Mit einer problemabhängigen Wahl der Konstante c könne die Auswirkung der beiden Potentialfunktionen gesteuert werden.

Dies ist leider im Allgemeinen nicht in der Intention von [GH04] möglich: Wird c positiv gewählt (die Autoren wählen einen konstanten Anteil von $\widehat{\mathrm{ed}}_s(z)$), so ist für alle Knoten um s herum zu erwarten, dass $\widehat{\mathrm{ed}}_s(\cdot)$ und $\widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot)$ jeweils ungefähr null sind, sodass dort – wie beabsichtigt – $\widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot)$ zum Einsatz kommt. In der Nähe von z ist $\widehat{\mathrm{ed}}_z$ ungefähr null und es wird $\widehat{\mathrm{ed}}_s(z)$ ungefähr gleich $\widehat{\mathrm{ed}}_z(s)$ sein, sodass auch hier – entgegen der Absicht – $\widehat{\mathrm{ed}}_z(s) - \widehat{\mathrm{ed}}_z(\cdot)$ benutzt wird. Die Argumentation trifft auch auf den kürzesten Weg von s nach z zu. Wählt man die Konstante $c < 0$, so ist zu erwarten, dass in dem zur Suche relevanten Bereich ausschließlich $\widehat{\mathrm{ed}}_s(\cdot)$ benutzt wird. Mit $c = 0$ werden sich die beiden Potentialfunktionen entlang des kürzesten Weges kaum unterscheiden.

Bewertung

Da in G_π die beidseitige Suche nach Dijkstra durchgeführt wird, sind zwei Kriterien zu beachten: Die Distanz von s und z in G_π im Vergleich zu der in G und inwieweit die Reduzierung der Kantengewichte zielgerichtet durchgeführt wird.

Die Distanzen werden in G_π bei den einzelnen Verfahren wie folgt reduziert:

- Verwendung nur einer Schätzfunktion:
  $\mathrm{dist}_\pi(s,z) = \mathrm{dist}(s,z) - \widehat{\mathrm{ed}}_z(s)$ bzw. $\mathrm{dist}_\pi(s,z) = \mathrm{dist}(s,z) - \widehat{\mathrm{ed}}_s(z)$
- Linearkombination beider Schätzfunktionen:
  $\mathrm{dist}_\pi(s,z) = \mathrm{dist}(s,z) - \alpha \cdot \widehat{\mathrm{ed}}_z(s) - (1-\alpha) \cdot \widehat{\mathrm{ed}}_s(z)$
- Maximumbildung nach [GH04]: Wie oben beschrieben kommt im Wesentlichen nur jeweils eine der Schätzfunktionen auf dem kürzesten Weg zum Einsatz, somit
  $\mathrm{dist}_\pi(s,z) = \mathrm{dist}(s,z) - \widehat{\mathrm{ed}}_z(s)$ bzw. $\mathrm{dist}_\pi(s,z) = \mathrm{dist}(s,z) - \widehat{\mathrm{ed}}_s(z)$

Somit gilt diesbzgl., dass die Verwendung nur einer Schätzfunktion die beste Wahl ist. Man wähle $\widehat{\mathrm{ed}}_z(\cdot)$ falls $\widehat{\mathrm{ed}}_z(s) > \widehat{\mathrm{ed}}_s(z)$ und $\widehat{\mathrm{ed}}_s(\cdot)$ sonst. Falls $\widehat{\mathrm{ed}}_z(s) = \widehat{\mathrm{ed}}_s(z)$ ist auch die Linearkombination gleich gut für beliebiges $\alpha,\ 0 \leq \alpha \leq 1$.

Bei Verwendung nur einer Schätzentfernung $\widehat{\mathrm{ed}}_z(\cdot)$ werden die Kantengewichte nahe des Startknotens in Richtung Zielknoten sehr effizient reduziert, in entgegengesetzter Richtung werden die Kantengewichte in G_π sogar größer. In der Nähe des Zielknotens hingegen werden alle umliegenden Kantengewichte gleichermaßen reduziert, sodass der am Zielknoten beginnende Dijkstra-Algorithmus in G_π nicht zielgerichtet zum Startknoten hin suchen wird. Wie oben beschrieben trifft dies auch auf die Maximumbildung in der Methode nach [GH04] zu. Nur jenseits des Start- bzw. Zielknotens kommt dann die „bessere“ Potentialfunktion zum Zug.

Bei der Linearkombination wird der Vorteil der zielgerichteten Gewichtsreduzierung sowohl am Start- wie auch am Zielknoten ausgenutzt, dies jedoch jeweils in abgeschwächter Form. Entlang des kürzesten Weges verstärkt sich die Reduzierung ähnlich wie bei Berücksichtigung nur einer Schätzfunktion.

Somit ist bei etwa gleich guten Schätzfunktionen $\widehat{\mathrm{ed}}_z(\cdot)$ und $\widehat{\mathrm{ed}}_s(\cdot)$ zu erwarten, dass die Anzahl der betrachteten Knoten bei der beidseitigen Dijkstra-Suche in G_π mit der Wahl $\alpha = \frac{1}{2}$ am geringsten ist. Je besser die Schätzfunktion $\widehat{\mathrm{ed}}_z(\cdot)$ gegenüber $\widehat{\mathrm{ed}}_s(\cdot)$ ist, umso stärker sollte sie durch geeignetes α berücksichtigt werden.

[GH04] selbst haben in Experimenten festgestellt, dass ihre Methode der von [IHI94] unterlegen ist, ohne dies jedoch erklären zu können.

Kapitel 5

Bucketstrukturen in Kürzeste-Wege-Algorithmen

Die in den Kapiteln 3 und 4 vorgestellten Verfahren sind alle nur vergleichsbasiert. Das heißt, dass aufgrund des Satzes 3.38 bei Verbesserungen der Laufzeit entweder die Knoten nicht in aufsteigender Reihenfolge der Entfernungen aufgenommen werden (müssen) oder der Algorithmus nicht ausschließlich auf Vergleichen basieren kann. Wir betrachten in diesem Abschnitt Algorithmen, die mit beschränkten Kantengewichten arbeiten. Für die Kantengewichte seien dabei $\gamma_{\min} := \min(\gamma(E))$ und $\gamma_{\max} := \max(\gamma(E))$.

Fast alle Algorithmen, die auf Graphen mit beschränkten Kantengewichten arbeiten, benutzen Bucket-Strukturen zur Verwaltung der Werte $\mathrm{D}(\cdot)$. Jeder Bucket wird dabei als doppelt-verkettete Liste verwaltet, sodass das Einfügen und Löschen in einem Bucket in konstanter Zeit möglich ist.

5.1 Dials Algorithmus

Der erste Kürzeste-Wege-Algorithmus, der Buckets benutzt, geht auf [Dia69] zurück. Er implementiert lediglich die in Dijkstras Algorithmus (3.30 mit Lemma 3.34) verwendeten Funktionen insert$(\cdot)$, decrease_key$(\cdot)$ und delete_min$(\cdot)$ unter Verwendung eines Bucketarrays A neu. Die Kantengewichte $\gamma : E \to \mathbb{N}_0$ sind ganzzahlig und positiv, ein Knoten v mit $\mathrm{D}(v) = i$ befindet sich dabei stets im Bucket $A[i]$. Die Prozeduren insert$(\cdot)$ und decrease_key$(\cdot)$ lassen sich so direkt in $O(1)$ implementieren. Für das delete_min$(\cdot)$ wird im Algorithmus in einer Variable l (initialisiert mit $l := 0$) der gerade betrachtete Index im Bucketarray A gemerkt. Das

delete_min(·) entnimmt – wenn möglich – Knoten aus $A[l]$. Ist der Bucket $A[l]$ leer, so wird $l := l + 1$ erhöht. Da in Dijkstras Algorithmus die Knoten in aufsteigender Entfernung zum Startknoten aufgenommen werden (Lemma 3.35), folgt direkt, dass die Buckets $A[i]$ mit $i < l$ stets leer sind, sodass l nie verringert werden muss.

Satz 5.1 (Dials Implementierung) *Dials Implementierung hat eine Laufzeit von* $O(m + n \cdot \gamma_{\max})$.

Beweis: Bevor alle Knoten in die Baummenge B aufgenommen wurden, müssen alle Buckets vom Index 0 bis $\max(\mathrm{d}(V))$ betrachtet worden sein. Der Aufwand für die delete_min(·)-Aufrufe beträgt somit $O(n+\max(\mathrm{d}(V)))$. Wie schon beim generischen Algorithmus (Satz 3.21) ist die Länge jedes doppelpunktfreien Weges und damit auch $\max(\mathrm{d}(V))$ durch $(n-1) \cdot \gamma_{\max}$ beschränkt. Der Aufwand der insert(·)- und decrease_key(·)-Aufrufe beträgt zusammen $O(m)$, womit die obige Gesamtlaufzeit bewiesen ist. □

Der größte auftrende D(·)-Wert ist durch $\max(\mathrm{d}(V))+\gamma_{\max}$ beschränkt.[1] Aufgrund folgender Eigenschaft des Dijkstra-Algorithmus reicht für das Bucketarray A eine deutlich kleinere Größe.

Lemma 5.2 (Intervalleigenschaft von D(·)**)** *Führt man den Algorithmus 3.30 mit Lemma 3.34 durch, so gilt* $\max(\mathrm{D}(R)) \leq \min(\mathrm{D}(R)) + \gamma_{\max}$.

Beweis: Aus der Invariante 3.29 und dem Korollar 3.36 folgt

$$\max(\mathrm{D}(R)) \leq \max(\mathrm{d}(B)) + \gamma_{\max} \leq \min(\mathrm{D}(R)) + \gamma_{\max}$$

□

Das Lemma 5.2 gilt offensichtlich nicht, wenn man die A^*-Heuristik (Lemma 3.40) verwendet.

Korollar 5.3 *In Dials Implementierung von Dijkstras Algorithmus können höchstens* $\gamma_{\max} + 1$ *Buckets gleichzeitig nichtleer sein. Die Indizes sind dabei aus dem Intervall* $[\min(\mathrm{D}(R)) \ldots \min(\mathrm{D}(R)) + \gamma_{\max}]$.

Es reicht demnach ein Bucketarray $A[0 \ldots \gamma_{\max}]$ aus. Die Buckets mit größerem Index i können auf $A[i \mod (\gamma_{\max} + 1)]$ abgebildet werden.

[1] Ein größerer Wert ist aufgrund der Invariante 3.29 nicht möglich. Der Wert wird in folgendem Beispiel auch angenommen: Sei v mit $\mathrm{d}(v) = \max(\mathrm{d}(V))$ und seien v' und v'' nur untereinander und mit v verbunden ($\gamma(vv') = 0$, $\gamma(vv'') = \gamma_{\max}$, $\gamma(v'v'') = 0$), so nimmt v'' nach Auswahl von $v \in R$ den Wert $\mathrm{D}(v'') = \max(\mathrm{d}(V)) + \gamma_{\max}$ an.

5.2 Multilevel-Bucket-Varianten

[DF79] und [CGR96] geben Modifizierungen von Dials Algorithmus an, die die Laufzeit und den Speicherbedarf deutlich senken.

Die Idee ist zunächst, gegenüber Dials Implementierung Platz zu sparen. Wir verwenden statt $\gamma_{\max}+1$ nur b Buckets für ein festes $b < \gamma_{\max}+1$. Der Algorithmus läuft in Phasen ab: In der i-ten Phase (beginnend mit $i=1$) umspannt das Bucketarray das Intervall $I_i := [(i-1)\cdot b \ldots i\cdot b - 1]$. Alle Knoten v mit $\mathrm{D}(v) \geq i\cdot b$ werden in einem gesonderten *Überlaufbucket* gespeichert. Erreicht der Index l den Wert $i\cdot b$, so ist nach Lemma 3.35 das gesamte Bucketarray leer und eine neue Phase $i+1$ mit dem Intervall I_{i+1} beginnt, indem die Knoten aus dem Überlaufbucket auf die entsprechenden Buckets in I_{i+1} verteilt bzw. wieder in einen neuen Überlaufbucket aufgenommen werden: Ein Knoten v mit $\mathrm{D}(v) < (i+1)\cdot b$ kommt in den Bucket $A[\mathrm{D}(v) - i\cdot b]$ $(= A[\mathrm{D}(v) \mod b])$, ist $\mathrm{D}(v) \geq (i+1)\cdot b$ kommt v wieder in den Überlaufbucket. Aufgrund des Korollars 5.3 kann ein Knoten sich in höchstens $O(\gamma_{\max}/b)$ Phasen im Überlaufbucket befinden, bevor er zum Baumknoten wird. Der Aufwand für die Verteilung auf die Buckets beträgt somit $O(n\cdot\gamma_{\max}/b)$. Der Gesamtaufwand beträgt trotz gesparten Platzaufwands weiterhin $O(m+n\cdot\gamma_{\max})$. Eine kleine Modifzierung erhält nicht nur den oben gesparten Platz, sondern verbessert auch deutlich die Laufzeit.

Satz 5.4 *Dials Algorithmus lässt sich mit $O(\sqrt{\gamma_{\max}})$ Platz und Laufzeit $O(m+n\cdot\sqrt{\gamma_{\max}})$ implementieren.*

Beweis: Das Intervall I_i umfasst nun die Werte $[\min_i \ldots \min_i + b - 1]$ mit $\min_1 = 0$. Erreicht der Index l den Wert $\min_i + b$, so befinden sich die Knoten aus R im Überlaufbucket. Setze $\min_{i+1} := \min(\mathrm{D}(R))$ auf den kleinsten $\mathrm{D}(\cdot)$-Wert im Überlaufbucket und führe den Algorithmus mit $l := \min_{i+1}$ fort. Der Aufwand der Minimumbestimmung vergrößert nur den Faktor im $O(n\cdot\gamma_{\max}/b)$ für das Verteilen der Knoten aus den Überlaufbuckets. Da in jeder Phase mindestens ein Knoten zum Baumknoten wird, nimmt der Index l nur $O(n\cdot b)$ Werte an. Die insert$(\cdot)$- und decrease_key$(\cdot)$-Aufrufe benötigen zusammen $O(m)$ Schritte. Der Gesamtaufwand beträgt damit $O(m+n\cdot(b+\gamma_{\max}/b))$. Mit der Wahl $b := \lfloor\sqrt{\gamma_{\max}}\rfloor$ folgt die Behauptung. □

Der Worst-Case wird z.B. in dem Graphen $G = (\{v_0, \ldots, v_{n-1}\}, E, \gamma)$ mit $E = \{v_{i-1}v_i \mid 0 < i < n\} \cup \{v_{i-c}v_i \mid c \leq i < n\}$, $\gamma(v_{i-1}v_i) = c$, $0 < i < n$ und $\gamma(v_{i-c}v_i) = c^2 =: \gamma_{\max}$, $c \leq i < n$ erreicht. Bis auf $v_0, \ldots, v_{c-1}$ sind alle Knoten $\Theta(c)$ mal im Überlaufbucket.[2]

Als Vorbereitung zur rekursiven Anwendung ist die Formulierung mit zwei Bucketarrays hilfreich – die Laufzeit und der Platzverbrauch bleiben dabei bzgl. $O(\cdot)$-Notation unverändert: Zu dem Parameter b (wir wählen später wieder $b \in \Theta(\sqrt{\gamma_{\max}})$) sei $A_0[0 \ldots b-1]$ ein Level-0-Bucketarray (jeder Bucket in diesem Array umfasst ein Intervall der Breite $b^0 = 1$) und $A_1[0 \ldots \lceil n \cdot \gamma_{\max}/b \rceil]$ ein Level-1-Bucketarray, in dem jedes $A_1[i]$ einem Intervall $I_i := [i \cdot b \ldots (i+1) \cdot b - 1]$ der Breite $b^1 = b$ entspricht. Zu jedem Zeitpunkt ist genau ein Level-1-Bucket $A_1[i]$ auf das Level-0-Bucketarray $A_0[0 \ldots b-1]$ verfeinert – zu Beginn ist dies $A_1[0]$. Die Buckets $A_1[i']$, $i' < i$ sind jeweils bereits leer. In $A_0[j]$ werden die Knoten mit $\mathrm{D}(\cdot) = i \cdot b + j$ abgelegt.

Wir merken uns den Index i_1 des gerade verfeinerten Level-1-Buckets und den Wert $t_0 := i_1 \cdot b + b - 1$, der den größten Wert darstellt, der im Bucketarray $A_0[\ldots]$ verwaltet wird. Wird der Wert eines Knotens v durch insert$(\cdot)$ oder decrease_key$(\cdot)$ auf den Wert $\mathrm{D}(v)$ verringert, so wird v aus seinem Bucket gelöscht und neu eingefügt: Ist $\mathrm{D}(v) > t_0$, so wird v in den Level-1-Bucket $A_1[\mathrm{D}(v) \text{ div } b]$ eingefügt – wird $\mathrm{D}(\cdot)$ als Zahl zur Basis b dargestellt, so entspricht dies dem Abschneiden der letzten Stelle – wählt man b als eine 2er-Potenz, so lassen sich die Bucket-Indizes auf einer RAM durch eine einfache Shift-Operationen berechnen. Ist $\mathrm{D}(v) \leq t_0$, so wird v in den Level-0-Bucket $A_0[\mathrm{D}(v) \mod b]$ eingefügt.

Sind alle Knoten aus $A_0[\ldots]$ entfernt, wird der nächste nichtleere Level-1-Bucket gesucht und verfeinert. Jeder Knoten v aus diesem Level-1-Bucket wird in $A_0[\mathrm{D}(v) \mod b]$ eingefügt.

Aufgrund des Korollars 5.3 können wir uns durch Modulobildung mit $\lceil \gamma_{\max}/b \rceil + 1$ auf ein Bucketarray $A_1[0 \ldots \lceil \gamma_{\max}/b \rceil]$ beschränken.

Gegenüber der Überlaufbucketvariante spart man sich die Suche nach dem Minimum $\min(\mathrm{D}(R))$ im Überlaufbucket, muss dafür aber ggf. einige Level-1-Buckets auf Leerheit testen (Aufwand je $O(1)$), bevor der nächste Level-1-Bucket auf $A_0[\ldots]$ verfeinert wird. Analog zur Überlaufbucketvariante können höchstens n Level-1-Buckets verfeinert werden, die Anzahl der

[2]Zum Erreichen des Worst-Case reicht es schon, wenn in jeder Phase nur ein Knoten zum Baumknoten wird, sodass der Index l in $n-1$ Phasen je b Werte annimmt, z.B. in einem zu einer Liste entarteten Graphen $G = (\{v_0, \ldots, v_{n-1}\}, \{v_{i-1}v_i \mid 0 < i < n\}, \gamma)$ mit $\gamma(\cdot) \equiv \gamma_{\max}$.

Level-1-Buckets ist durch $O(n \cdot \gamma_{\max}/b)$ beschränkt. Der Speicherbedarf hat sich gegenüber der Überlaufbucketvariante im Wesentlichen verdoppelt.

Die Verallgemeinerung auf k Level findet sich in einigen Varianten ([DF79] (dort noch mit schlechteren Laufzeitabschätzungen), [AMOT90], [CGR96], [CGS99], [Gol04]), die sich hauptsächlich im Detail in der Verwaltung der Bucketintervalle unterscheiden, Laufzeit und Speicherverbrauch sind gleich. Durch die Verteilung auf mehrere Level – wir werden $O(\log \gamma_{\max}/\log\log \gamma_{\max})$ Level benutzen – werden nochmals weniger Buckets benötigt ($O((\log \gamma_{\max}/\log\log \gamma_{\max})^2)$). Die Funktionen insert$(\cdot)$ und delete_min$(\cdot)$ sind so realisiert, dass ein Knoten jedes Mal, wenn er angefasst wird, in ein tieferes Level rutscht und so pro Knoten nur ein Aufwand von $O(\log \gamma_{\max}/\log\log \gamma_{\max})$ Schritten nötig ist. Der Aufwand der delete_min$(\cdot)$-Aufrufe kann ebenfalls über alle Knoten amortisiert werden. Zusätzlich zu dem über die Knoten amortisierten Aufwand haben insert$(\cdot)$ und decrease_key$(\cdot)$ nur konstanten Aufwand und delete_min$(\cdot)$ den Aufwand $O(\log \gamma_{\max}/\log\log \gamma_{\max})$, sodass eine Gesamtlaufzeit von $O(m + n \cdot \log \gamma_{\max}/\log\log \gamma_{\max})$ erreicht wird.

Der Ablauf des Algorithmus, also der Verlauf der Werte D$(\cdot)$ und die Reihenfolge, in der die Knoten in B aufgenommen werden, ist auch hier vollständig identisch zu dem des Dijkstra-Algorithmus, sodass die Korrektheit und Terminierung direkt aus Satz 3.31 und Lemma 3.34 folgt. Der einzige Unterschied besteht in der Verwaltung der Werte D(R) mittels der Multilevel-Bucket-Struktur.

Wir versuchen hier, die Vorzüge der einzelnen Darstellungen zu vereinen und die in den Artikeln stark verkürzt dargestellten Passagen auszuführen.

Wird der Parameter $b < \gamma_{\max} + 1$ gewählt, so werden k Level, $b^k \geq \gamma_{\max} + 1$, benötigt (wir setzen $k := \lceil \log_b(\gamma_{\max} + 1) \rceil$). Auf jedem Level i, $0 \leq i < k$ verwenden wir ein Bucketarray $A_i[0 \ldots b-1]$, wobei jeder Bucket ein Intervall der Breite b^i umfasst und das Array $A_i[\ldots]$ genau einen Bucket aus $A_{i+1}[\ldots]$ verfeinert. Formal gibt es zusätzlich noch ein Top-Level-Bucket $A_k[0 \ldots n]$, das Buckets der Breite b^k enthält. Von diesen können aufgrund des Korollars 5.3 nur zwei gleichzeitig belegt sein, von denen zusätzlich einer in dem Bucketarray $A_{k-1}[\ldots]$ verfeinert ist. Es reicht also ein Überlaufbucket wie in Satz 5.4.[3]

Die Verteilung der Knoten auf die Buckets der einzelnen Level beruht auf der Zahldarstellung des zuletzt durch delete_min$(\cdot)$ gewählten Minimums $D_{\min}$.

[3]Zugunsten der einfacheren Darstellung wandert hier der Überlaufbucket ebenfalls durch ein Bucketarray $A_k[0 \ldots b-1]$ mit b Buckets. Bei einem expliziten Überlaufbucket wären zusätzliche **if**-Abfragen nötig, wenn Knoten auf Level k betrachtet werden.

Ist $\mathrm{D}_{\min} := \sum_{i=0}^{\infty} b_i \cdot b^i$ zur Basis b dargestellt, $\mathrm{D}_{\min} = b_\infty \cdots b_k b_{k-1} \cdots b_1 b_0$, so geben die Zahlen $(b_\infty \cdots b_k b_{k-1} \cdots b_{i+2} b_{i+1}) \cdot b^{i+1}$ die jeweils kleinsten auf dem Level i verwalteten Werte an.

Zu jedem Level i, $0 \leq i \leq k$ merken wir uns die Gesamtanzahl der Knoten n_i auf dem Level i (ohne die Knoten aus dem jeweils gerade verfeinerten Bucket) und t_i, $0 \leq i < k$, das den größten Wert repräsentiert, der zum betreffenden Zeitpunkt im Bucketarray $A_i[\ldots]$ verwaltet werden kann. Es gilt $t_i := (b_\infty \cdots b_k b_{k-1} \cdots b_{i+2} b_{i+1}) \cdot b^{i+1} + b^{i+1} - 1$ (d.h., dass in t_i die Stellen $b_i, \ldots, b_0$ jeweils den Wert $b-1$ haben). Die Werte t_i werden mit $t_i := b^{i+1} - 1$ initialisiert und später ausschließlich während der delete_min$(\cdot)$-Operationen auf Basis des $\mathrm{D}_{\min}$-Wertes neu gesetzt. Es gilt zu Beginn offensichtlich
$t_0 \leq t_1 \leq \cdots \leq t_{k-1}$. Diese Invariante wird stets aufrecht erhalten, da sich t_i und t_{i-1} höchstens in b_i unterscheiden können und in t_i diese Ziffer mit $b-1$ den größtmöglichen Wert annimmt.

Jeder Knoten wandert im Laufe des Algorithmus durch die Level in absteigender Reihenfolge der Indizes. Wird ein Knoten v mit $\mathrm{D}(v)$ durch insert$(\cdot)$ eingefügt, so wird durch Vergleich von $\mathrm{D}(v)$ mit den Werten t_i das Level mit kleinstem Index gefunden, für das $\mathrm{D}(v) \leq t_i$, und v dort eingefügt. Zur Vermeidung von zusätzlichen Indexvergleichen wird als „Wächter“ der Wert $t_{-1} := -1$ gesetzt.

1 $i := k$; **while** $\mathrm{D}(v) \leq t_{i-1}$ **do** $i := i - 1$ **od**;
2 $j := (\mathrm{D}(v)$ div $b^i)$ mod b;
3 Füge v in den Bucket $A_i[j]$ ein; $n_i := n_i + 1$

Man beachte: Ist der Bucket $A_{i+1}[b-1]$ auf das Array $A_i[\ldots]$ verfeinert, so ist $t_{i+1} = t_i$ und $\mathrm{D}(v)$ wird korrekt in $A_i[\ldots]$ eingefügt. Der Wert j entspricht in der Darstellung des $\mathrm{D}(v)$ zur Basis b der Ziffer b_i mit Wertigkeit b^i.

Lemma 5.5 *Ist* $\mathrm{D}_{\min} = b_\infty \cdots b_k b_{k-1} \cdots b_1 b_0$*, so sind für alle* i *die Buckets* $A_i[j]$, $0 \leq j < b_i$ *stets leer.*

Beweis: Für alle Knoten v auf Level i gilt

$$\mathrm{D}_{\min} \leq \mathrm{D}(v) \leq t_i = (b_\infty \cdots b_k b_{k-1} \cdots b_{i+2} b_{i+1}) \cdot b^{i+1} + b^{i+1} - 1$$

Insbesondere unterscheiden sich die Werte $\mathrm{D}(\cdot)$ auf Level i höchstens in den Ziffern $b_{i'}$, $i' \leq i$, sodass die Ziffer mit Wertigkeit b^i nicht kleiner als die entsprechende Ziffer von $\mathrm{D}_{\min}$ sein kann, da $\mathrm{D}_{\min} \leq \mathrm{D}(\cdot)$. □

Ist $i \neq 0$, so ist auch $A_i[b_i]$ leer. Wäre ein Knoten $v \in A_i[b_i]$, so würden sich $\mathrm{D}_{\min}$ und $\mathrm{D}(v)$ in der i-ten und höherwertigeren Stellen nicht unterscheiden und somit $\mathrm{D}(v) \leq t_{i-1}$ im Widerspruch zu $v \in A_i[\cdot]$. Die Knoten v mit $\mathrm{D}(v) = \mathrm{D}_{\min}$ befinden sich alle in $A_0[b_0]$.

Das decrease_key$(\cdot)$ wird bis auf die Initialisierung des i genau wie das insert$(\cdot)$ durchgeführt: Galt zuvor $\mathrm{D}(v) \in A_{i'}[j]$, so wird v aus $A_{i'}[j]$ gelöscht und dann wie bei insert$(\cdot)$ eingefügt, wobei die Suche nach dem richtigen Level mit dem Index $i := i'$ begonnen wird. Da sich der $\mathrm{D}(v)$-Wert durch das decrease_key$(\cdot)$ verringert, kann der Knoten v nicht in einen Bucket auf höherem Level verschoben werden.

1 Entferne v aus dem Bucket $A_{i'}[j]$; $n_{i'} := n_{i'} - 1$;
2 $i := i'$; **while** $\mathrm{D}(v) \leq t_{i-1}$ **do** $i := i - 1$ **od**;
3 $j := (\mathrm{D}(v) \text{ div } b^i) \mod b$;
4 Füge v in den Bucket $A_i[j]$ ein; $n_i := n_i + 1$

Der Gesamtaufwand in den Prozeduren insert$(\cdot)$ und decrease_key$(\cdot)$ ist somit $O(k)$ pro Knoten plus konstanten Aufwand je Aufruf (dies beinhaltet auch den Aufwand, falls ein Knoten durch ein decrease_key$(\cdot)$ auf demselben Level bleibt), insgesamt also $O(n \cdot k + m)$ Schritte.

Wird b als 2er-Potenz gewählt, so lassen sich die Modulo- und Divisionsoperationen auf einer RAM effizient durch Bit-Maskierung und Shift-Operationen durch je eine AC^0-Operation in $O(1)$ durchführen, insbesondere sind beliebige Shifts in konstanter Zeit möglich. Die nötigen Bitmasken können vorab leicht in $O(k)$ tabelliert werden und erhöhen nicht den asymptotischen Aufwand.

Es bleibt der Aufwand für die delete_min$(\cdot)$-Aufrufe: Es wird über die Anzahl n_i der Knoten pro Level zunächst das niedrigste nichtleere Level i in $O(k)$ bestimmt. Innerhalb des Levels i suchen wir in $O(b)$ den ersten nichtleeren Bucket j. Aufgrund des Lemmas 5.5 kann die Suche mit $j := b_i + 1$ (bzw. $j := b_i$ für $i = 0$) begonnen werden, wobei b_i der i-ten Stelle des $\mathrm{D}_{\min}$ entspricht. War $i = 0$, so haben wegen der ganzzahligen Kantengewichte alle Knoten des Buckets $A_0[j]$ denselben Wert und wir können einen beliebigen dieser Knoten als Ergebnis des delete_min$(\cdot)$ zurückgeben, die t_i bleiben alle unverändert. Ansonsten bestimmt das Element mit dem kleinsten Wert $\mathrm{D}(v)$ wie oben beschrieben die neuen Werte $t_{i'}$ für $i' < i$. Da sich v auf Level i befindet, unterscheidet sich das bisherige $\mathrm{D}_{\min}$ von $\mathrm{D}(v)$ nicht in den Stellen mit Wertigkeit größer b^i, sodass die Werte $t_{i'},\ i' \geq i$ unverändert bleiben. Die Knoten des Buckets $A_i[j]$ werden dann analog zum decrease_key$(\cdot)$ verschoben.

1 $i := 0$; **while** $n_i = 0$ **do** $i := i + 1$ **od**;
2 **if** $i = 0$ **then**
2a.1 $j := \mathrm{D}_{\min} \mod b$; **while** $A_0[j] = \emptyset$ **do** $j := j + 1$ **od**;
2a.2 wähle $u \in A_0[j]$; $\mathrm{D}_{\min} := \mathrm{D}(u)$;
2a.3 Entferne u aus dem Bucket $A_0[j]$; $n_0 := n_0 - 1$
2 **else**
2b.1 $j := ((\mathrm{D}_{\min} \operatorname{div} b^i) \mod b) + 1$; **while** $A_i[j] = \emptyset$ **do** $j := j + 1$ **od**;
2b.2 $\mathrm{D}_{\min} := \min(A_i[j])$; wähle $u \in A_i[j]$ mit $\mathrm{D}(u) := \mathrm{D}_{\min}$;
2b.3 Entferne u aus dem Bucket $A_i[j]$; $n_i := n_i - 1$;
2b.4 **for** $i' := 0$ **to** $i - 1$ **do** $t_{i'} := (\mathrm{D}_{\min} \operatorname{div} b^{i'+1}) \cdot b^{i'+1} + b^{i'+1} - 1$ **od**;
2b.5 **for all** $v \in A_i[j]$ **do**
2b.5.1 Entferne v aus dem Bucket $A_i[j]$; $n_i := n_i - 1$;
2b.5.2 $i' := i - 1$; **while** $\mathrm{D}(v) \leq t_{i'-1}$ **do** $i' := i' - 1$ **od**;
2b.5.3 $j' := (\mathrm{D}(v) \operatorname{div} b^{i'}) \mod b$;
2b.5.4 Füge v in den Bucket $A_{i'}[j']$ ein; $n_{i'} := n_{i'} + 1$
2b.5 **od**
2 **fi**;
3 **return** u

Ist das neue $\mathrm{D}_{\min} = b_\infty \cdots b_k b_{k-1} \cdots b_1 b_0$, so gilt für alle Knoten v aus dem zu verfeinernden Bucket $A_i[j]$ (falls $i \neq 0$)

$$\mathrm{D}_{\min} \leq \mathrm{D}(v) \leq t_{i-1} = (b_\infty \cdots b_k b_{k-1} \cdots b_{i+1} b_i) \cdot b^i + b^i - 1$$

(für alle $\mathrm{D}(v)$ ist $b_i = j$), sodass jedesmal, wenn ein Knoten in einem delete_min($\cdot$)-Aufruf betrachtet wird, dieser auf ein echt kleineres Level verteilt wird – mindestens von Level i auf Level $i - 1$. Der nötige Aufwand pro Knoten ist daher $O(k)$, die Schritte der Minimumsuche in Schritt 2b.2 amortisieren sich gleichermaßen zu $O(k)$ pro Knoten. Der Aufwand für alle delete_min($\cdot$)-Aufrufe ist damit $O(n(k + b))$ Schritte.

Satz 5.6 (Multilevel-Bucket-Algorithmus) *Mit der Multilevel-Bucket-Struktur lässt sich Dijkstras Algorithmus mit* $O((\log \gamma_{\max} / \log\log \gamma_{\max})^2)$ *Platz und* $O(m + n \cdot \log \gamma_{\max} / \log\log \gamma_{\max})$ *Laufzeit implementieren.*

Beweis: Der Platzaufwand beträgt $O(k \cdot b)$, die Laufzeit beträgt $O(m + n(k + b))$. Mit der Wahl $b := \lceil \log \gamma_{\max} / \log\log \gamma_{\max} \rceil$ ist

$$\begin{aligned} k \in \Theta(\log_b \gamma_{\max}) &= \Theta(\log \gamma_{\max} / \log b) \\ &= \Theta(\log \gamma_{\max} / (\log\log \gamma_{\max} - \log\log\log \gamma_{\max})) = \Theta(b) \end{aligned}$$

wodurch sich die behaupteten Schranken ergeben. □

Ist $\log \gamma_{\max} / \log\log \gamma_{\max} \in \omega(n)$, so ist die Initialisierung der Bucketarrays in der angegebenen Laufzeit nicht mehr möglich. Man kann sich nun entweder darauf beschränken, nur die benötigten Indizes mit Universal Hashing zu verwalten ([CW79],[DHKP97]), sinnvoller (und einfacher) ist es jedoch, in diesem Fall den Dijkstra-Algorithmus zu verwenden. Bei $\log \gamma_{\max} / \log\log \gamma_{\max} \in \omega(n)$ ist selbst die einfache Listenimplementierung des Dijkstra-Algorithmus mit $O(n^2)$ schneller als die Multilevel-Bucket-Variante nach Satz 5.6.

Es folgt nun noch ein kurzer Vergleich zwischen den Varianten in der Darstellung: In [DF79] ist im Wesentlichen nur die Idee der Multilevel-Bucket-Struktur formuliert. Die Darstellung ist dort noch umständlich, unklar und sehr verkürzt, ebenso sind die Laufzeitabschätzungen noch schlechter.

[AMOT90] baut nicht auf der Zahldarstellung zur Basis b auf, verwendet aber ebenfalls Buckets der Breite b^i auf Level i. Genauer gesagt wird dort ein Bucketarray verwendet, in dem der i-te Bucket die Breite b^i hat und dieser in b Segmente unterteilt ist. [AMOT90] nennen dies einen *Two-Level-Radix-Heap*. Der Bucket $A_i[j]$ in obiger Darstellung entspricht im Two-Level-Radix-Heap dem j-ten Segment im i-ten Bucket. Im delete_min$(\cdot)$ wird dann stets ein Segment auf Buckets mit niedrigerem Index verteilt. Die Verwaltung der Bucket- und Segmentgrenzen ist aufwendiger, die Bucket- und Segmentgrößen verändern sich dynamisch und es ist mehr Beweistechnik nötig, um zu zeigen, dass sich Segmente stets vollständig auf Buckets mit kleinerem Index verteilen, sodass die Laufzeit aus Satz 5.6 erreicht wird.

In [CGR96], [CGS99] und [Gol04], die ausschließlich auf der Zahldarstellung zur Basis b aufbauen, wird hingegen der Ablauf des Algorithmus mit dem Wandern der Knoten durch die Level bei Weitem nicht so deutlich wie in [AMOT90].

5.3 Die Verbesserung nach [AMOT90] durch Verwaltung der Bucketindizes in einem Fibonacci-Heap

Allein die Suche nach dem Bucket, der den kleinsten D$(\cdot)$-Wert enthält, dominiert (bis auf das $O(m)$, das aber unvermeidbar ist, da jede Kante wenigstens einmal betrachtet werden muss) die Laufzeit beim Multilevel-Bucket-Ansatz (Satz 5.6). Alle anderen Operationen tragen nur zur einer Vergrößerung der Konstante im $O(\cdot)$ bei. [AMOT90] schlagen vor, statt nach diesem Bucket

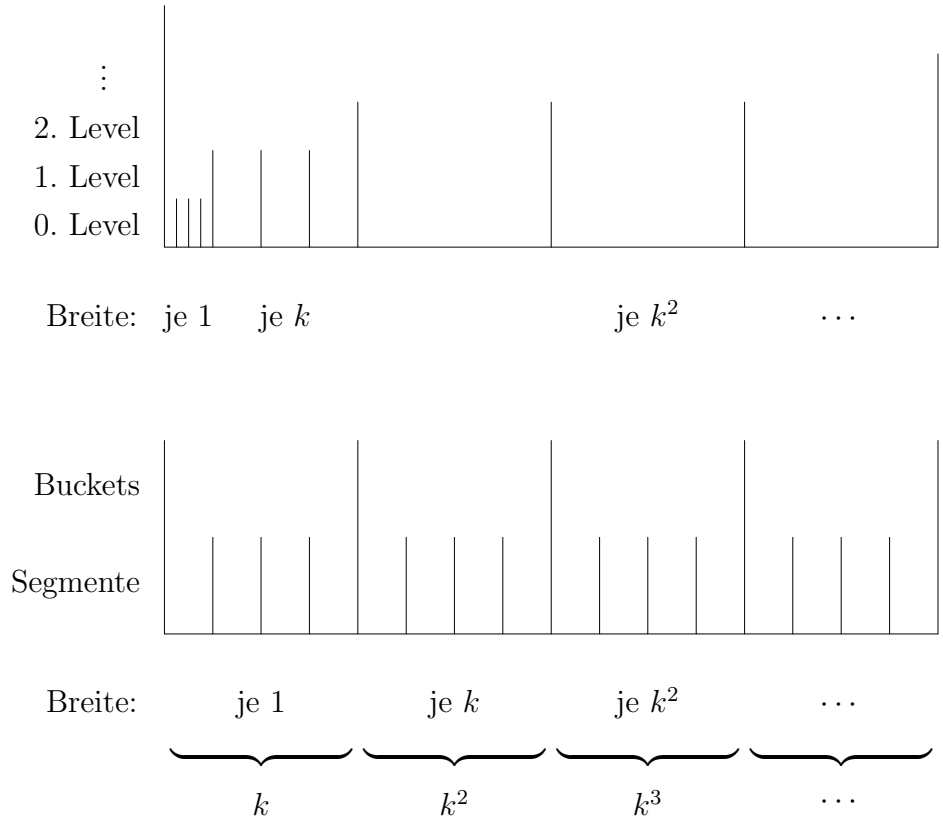


Abbildung 5.1: Vergleich der Multilevel-Bucket-Struktur ([DF79], [CGR96], [CGS99], [Gol04]) mit dem Two-Level-Radix-Heap ([AMOT90])

explizit zu suchen, mit einer angepassten Variante der Fibonacci-Heaps die Indizes der Buckets zu verwalten, in denen sich die Knoten der Menge R befinden. Die Indizes lassen sich als Paare von Level l und Index i auf dem Level l beschreiben, wobei $(l, i) < (l', i')$ gilt, wenn $l < l'$ ist oder $l = l'$ und $i < i'$. Die angepasste Variante der Fibonacci-Heaps hat wie normale Fibonacci-Heaps für die insert(·)- und decrease_key(·)-Operationen einen amortisierten Aufwand $O(1)$. Indizes, die mehrfach vorkommen, werden nur durch einen Repräsentanten im Heap gespeichert, sodass sich zu jedem Zeitpunkt nur so viele Elemente in dem Fibonacci-Heap befinden, wie es verschiedene Indizes der Knoten in R gibt. Diese Zahl ist durch $O(\min\{n, k \cdot b\})$ beschränkt. Der amortisierte Aufwand der delete_min(·)-Operation beträgt $O(\log(\min\{n, k \cdot b\}))$. Jedes insert(·) und decrease_key(·) im Multilevel-Bucket benötigt auch ein insert(·) bzw. decrease_key(·) für die entsprechenden Indizes im Fibonacci-Heap. Das delete_min(·) im Multilevel-Bucket zieht neben einem delete_min(·) im Fibonacci-Heap, um den ersten

nichtleeren Bucket zu finden, für jeden Knoten ein insert(·) nach sich: Da die Knoten des Buckets verteilt werden, müssen diese mit neuem Index wieder in den Fibonacci-Heap eingefügt werden; ggf. werden sie nur in die Menge der Knoten aufgenommen, die bereits durch einen Knoten mit gleichem Level-Index-Paar im Fibonacci-Heap repräsentiert wird. D.h., dass durch die n delete_min(·)-Aufrufe im Multilevel-Bucket bis zu $n \cdot k$ Aufrufe im Fibonacci-Heap mit je $O(1)$ Aufwand nötig werden. Der amortisierte Aufwand der so modifizierten Datenstruktur ist je $O(1)$ für insert(·)- und decrease_key(·) und $O(\log(\min\{n, k \cdot b\}) + k) \subseteq O(k + \log b)$ für das delete_min(·). Die Änderung bewirkt also eine Verringerung des amortisierten Aufwandes des delete_min(·) im Multilevel-Bucket von $O(k + b)$ auf $O(k + \log b)$. Die im $O(\cdot)$ versteckte Konstante ist jedoch deutlich größer.

Satz 5.7 *Mit der Multilevel-Bucket-Struktur erweitert nach [AMOT90] lässt sich Dijkstras Algorithmus mit $O(m + n \cdot \sqrt{\log \gamma_{\max}})$ Laufzeit implementieren.*

Beweis: Die Laufzeit beträgt $O(m + n(k + \log b))$ Schritte. Mit $b := 2^{\lceil \sqrt{\log \gamma_{\max}} \rceil}$ benötigen wir

$$k := \lceil \log_b(\gamma_{\max} + 1) \rceil \in O((\log \gamma_{\max})/(\log 2^{\sqrt{\log \gamma_{\max}}})) = O(\sqrt{\log \gamma_{\max}})$$

Level. Da $\log b \in O(\sqrt{\log \gamma_{\max}})$ können wir die Laufzeit von $O(m + n \cdot \sqrt{\log \gamma_{\max}})$ Schritten garantieren. □

Ist $k \cdot b \in \Omega(n)$, so kann man statt der angepassten Variante auch die mit weniger Aufwand verbundenen normalen Fibonacci-Heaps zur Verwaltung der Indizes benutzen. Dies führt zu einer etwas kleineren Konstante im $O(\cdot)$ der Laufzeit. Es ergibt sich jedoch wie oben ein Initialisierungsproblem: Der Platzbedarf $O(k \cdot b)$ ist mit der obigen Wahl der Parameter größer als in Satz 5.6. Ist $b \in \omega(n)$, so ist auch hier die Initialisierung der Bucketarrays nicht mehr in der angegebenen Laufzeit möglich, sodass wie oben Hashing oder andere Techniken zur Verwaltung der tatsächlich verwendeten Buckets benutzt werden müssten. Ist mit der obigen Wahl $b := 2^{\lceil \sqrt{\log \gamma_{\max}} \rceil} \in \omega(n)$, so gilt aber auch $\log n \in o(\sqrt{\log \gamma_{\max}})$, sodass in diesem Fall $O(m + n \cdot \log n) \subseteq O(m + n \cdot \sqrt{\log \gamma_{\max}})$ gilt und der Dijkstra-Algorithmus mit Fibonacci-Heaps verwendet werden kann, um die Laufzeit aus Satz 5.7 garantieren zu können.

Wir geben hier noch eine Alternative zu der Fibonacci-Heap-Variante, um die Laufzeit aus Satz 5.7 zu erreichen. Ausgangspunkt ist der Multilevel-Bucket-Algorithmus aus Satz 5.6 mit der Wahl von b und k wie im Satz 5.7. Wir

speichern zu jedem Bucketarray eine Zahl, deren Bits angeben, ob der entsprechende Bucket belegt ist oder nicht. Beim Einfügen des ersten Elements in einen Bucket und Löschen des letzten Elements muss dieses Bit gesetzt bzw. gelöscht werden, was z.B. durch Addition bzw. Subtraktion der entsprechenden 2er-Potenz in $O(1)$ durchgeführt werden kann. Im RAM-Modell kann der Wert auch einfach durch entsprechende Bitmasken und logische Operatoren gesetzt bzw. gelöscht werden. Sind innerhalb eines Bucketarrays den Buckets die Bits in absteigender Wertigkeit zugeordnet (der letzte Bucket habe Wertigkeit 2^0, der vorletzte 2^1 usw.), so lässt sich mittels binärer Suche leicht der Index des ersten nichtleeren Buckets in diesem Bucketarray bestimmen. Hat der erste nichtleere Bucket in dem Bitarray die Wertigkeit 2^i, so ist keines der höherwertigen Bits gesetzt und die zum Bucketarray gespeicherte Zahl liegt zwischen 2^i und $2^{i+1} - 1$, sodass binäre Suche benutzt werden kann (zunächst wird mit $2^{b/2}$ verglichen, danach mit $2^{b/4}$ bzw. $2^{3b/4}$ usw.). Das erste nichtleere Bucketarray lässt sich in $O(k)$ Schritten durch Betrachten der für die Bucketarrays $A_i[\ldots]$ gespeicherten Zahlen n_i bestimmen. Ist dieses gefunden, so findet man in $O(\log b)$ den gesuchten Bucket. Der amortisierte Aufwand des delete_min$(\cdot)$ ist somit wie oben $O(k + \log b)$, wodurch die Gesamtlaufzeit $O(m + n \cdot \sqrt{\log \gamma_{\max}})$ garantiert wird.

Dieser Ansatz entspricht dem normalen RAM-Modell jedoch nur, wenn $b \in O(\log n + \log \gamma_{\max})$ ist, mehr Bits stehen pro Speicherplatz nicht zur Verfügung. Für die Praxis sind die Werte aber noch klein genug, um gegenüber der modifizierten Fibonacci-Heap-Variante Vorteile zu bringen. Bei $\gamma_{\max} = 2^{64} - 1$ sind z.B. nur 256 Bits für jedes der 8 Bucketarrays zu speichern.

5.4 Das Kaliberlemma und die Algorithmen von Dinitz und Goldberg

Die oben dargestellten Varianten nehmen – da nur die insert$(\cdot)$-, decrease_key$(\cdot)$- und delete_min$(\cdot)$-Operationen neu implementiert wurden – als Implementierungsvariante des Dijkstra-Algorithmus weiterhin die Knoten in aufsteigender Reihenfolge der Entfernung zum Startknoten in die Menge B auf.

Aufbauend auf [Din78] und [Tho99] gibt [Gol04] ein weiteres Verbesserungspotential mit folgendem Lemma an, das für Knoten mit $\mathrm{D}(\cdot) > \min(\mathrm{D}(R))$ eine in $O(1)$ überprüfbare hinreichende Bedingung für $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ zur Verfügung stellt.

Lemma 5.8 (Kaliberlemma) *Seien $D_{\min}$ eine untere Schranke für $\min(\mathrm{D}(R))$ und $\forall v \in V: \ \gamma_{\min}(v) := \min\{\gamma(uv) \mid uv \in E\}$ die jeweils kleinsten Kantengewichte der in v mündenen Kanten. Ist $\mathrm{D}(v) \leq D_{\min} + \gamma_{\min}(v)$, so ist $\mathrm{D}(v) = \mathrm{d}(v)$.*

Beweis: $\mathrm{D}(v)$ kann sich nur durch ein decrease_key$(\cdot)$ verringern, es müsste also ein Knoten $u \in R$ existieren mit $\mathrm{D}(u) + \gamma(uv) < \mathrm{D}(v)$. Mit

$$\mathrm{D}(v) \leq D_{\min} + \gamma_{\min}(v) \leq \mathrm{D}(u) + \gamma(uv)$$

kann solch eine verkürzende Kante uv aber nicht mehr existieren und $\mathrm{D}(v)$ gibt bereits den korrekten Wert $\mathrm{d}(v)$ an. □

Die Aussage lässt sich noch etwas stärker formulieren, indem man nur die noch nicht betrachteten Eingangskanten zur Berechnung von $\gamma_{\min}(v) := \min\{\gamma(uv) \mid uv \in E, \ u \in R\}$ heranzieht. Dadurch wären die $\gamma_{\min}(v)$ jedoch vom Stand der Kürzeste-Wege-Berechnung abhängig und könnten nicht vorab in $O(n+m)$ berechnet werden. Der zusätzliche Aufwand in der Anwendung wäre größer als der zu erwartende Zeitgewinn.

Kombiniert man den Multilevel-Bucket-Ansatz mit dem Kaliberlemma, so zeigt [Gol04], dass mit der Wahl von $b = 2$ (oder jeder anderen Konstante), bei gleichverteilten Kantengewichten mit hoher Wahrscheinlichkeit lineare Laufzeit erreicht wird. Auch der Erwartungswert für die Laufzeit ist linear. In Experimenten mit sehr verschiedenen Graphenklassen vergleicht [Gol04] die Laufzeit mit der einfachen Breitensuche (Laufzeit $\Theta(n+m)$) auf den zugehörigen ungewichteten Graphen. Bei fast allen untersuchten Graphen ist die Laufzeit nur um einen Faktor von 1.5 bis 1.8 langsamer als die Breitensuche (unabhängig von der Knotenanzahl, die in den Experimenten bis zu 2 Millionen betrug). Bei keinem Graphen war der Faktor größer als 2.8, sodass für praktisch relevante Verbesserungen kaum noch Raum ist.

Der Mutlilevel-Bucket-Ansatz mit $b = 2$ lässt sich etwas einfacher darstellen, da hier auf jedem Level einer der je zwei Buckets verfeinert ist, somit also letztendlich nur k Buckets unterschiedlicher Breite benutzt werden – statt eines Bucketarrays auf Level i wird nur noch ein einzelner Bucket für dieses Level benötigt. Nur auf dem Level 0 werden beide Buckets des Arrays $A_0[\ldots]$ benötigt – der Bucket $A_0[1]$ wird zum Bucket mit Index 0. Statt des Buckets $A_0[0]$ verwalten wir eine Menge *Next*, die Knoten mit $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ enthält.[4]

[4]Es entsteht eine Datenstruktur ähnlich wie in [AMOT90] – dort *One-Level-Radix-Heap* genannt. Dieses ist ein Bucketarray, bei dem die Buckets mit Index 0 und 1 jeweils Breite 1 haben, Buckets mit Index i haben Breite 2^{i-1}. Die Knoten wandern im Laufe des Algorithmus hier durch das Bucketarray, wobei sich Knoten nach einem delete_min$(\cdot)$ stets auf Buckets mit echt kleinerem Index verteilen.

Solange *Next* nicht leer ist, entnimmt das delete_min(·) dieser Menge einen Knoten, nur sonst wird auf der Multilevel-Bucket-Struktur gearbeitet. Ebenso wie oben werden die Zahlen n_i und t_i verwaltet. Innerhalb eines Levels gibt es nun aber nur noch einen Bucket, sodass sämtliche Modulo-Berechnungen entfallen können. Das $\mathrm{D}_{\min}$ gibt den Wert des zuletzt aus dem Multilevel-Bucket entnommenen Knotens an und ist damit eine untere Schranke für $\min(\mathrm{D}(R))$ – die aus der Menge *Next* entnommenen Knoten können größere Werte $\mathrm{D}(\cdot)$ haben.

Das Kaliberlemma kann nun dazu benutzt werden, bei einer insert(·)− oder decrease_key(·)-Operation oder beim Neuverteilen der Knoten nach einer delete_min(·)-Operation jeweils zu entscheiden, ob für den Knoten bereits $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ garantiert werden kann – wenn ja, wird dieser in die Menge *Next* aufgenommen, sonst im Multilevel-Bucket neu eingefügt.

Aufgrund des Kaliberlemmas bleiben die niedrigen Level des Multilevel-Buckets frei – dies gilt unabhängig von der Wahl des Parameters b.

Lemma 5.9 *Die Level i mit $i < \lfloor \log_b \gamma_{\min} \rfloor$ des Mutlilevel-Buckets sind stets leer.*

Beweis: Angenommen, ein Knoten v würde in das Level $i \leq \lfloor \log_b \gamma_{\min} \rfloor - 1$ eingefügt, so ist

$$\mathrm{D}(v) \leq (\mathrm{D}_{\min} \text{ div } b^{i+1}) \cdot b^{i+1} + b^{i+1} - 1 < \mathrm{D}_{\min} + b^{i+1}$$

(vgl. Lemma 5.5), dann gilt mit $i \leq \lfloor \log_b \gamma_{\min} \rfloor - 1$ aber auch

$$\mathrm{D}(v) \leq \mathrm{D}_{\min} + b^{\lfloor \log_b \gamma_{\min} \rfloor} \leq \mathrm{D}_{\min} + \gamma_{\min} \leq \mathrm{D}_{\min} + \gamma_{\min}(v)$$

und nach Lemma 5.8 ist $\mathrm{D}(v) = \mathrm{d}(v)$. Der Algorithmus würde v in *Next* statt im Multilevel-Bucket einfügen. □

D.h., jeder Knoten wandert nur noch durch $O(\log \gamma_{\max} - \log \gamma_{\min}))$ Level. Somit sinkt auch der amortisierte Aufwand für die delete_min(·)-Operation.

Satz 5.10 *Mit der Multilevel-Bucket-Struktur und dem Kaliberlemma lässt sich Dijkstras Algorithmus mit $O((\log(\gamma_{\max}/\gamma_{\min})/\log\log(\gamma_{\max}/\gamma_{\min}))^2)$ Platz und $O(m + n \cdot \log(\gamma_{\max}/\gamma_{\min})/\log\log(\gamma_{\max}/\gamma_{\min}))$ Laufzeit implementieren.*

Beweis: Nach Lemma 5.9 werden nur

$$k' \in \Theta(k - \log_b \gamma_{\min}) = \Theta(\log_b(\gamma_{\max}/\gamma_{\min}))$$

Level benutzt. Dadurch reduziert sich der Platzaufwand gegenüber Satz 5.6 auf $O(k' \cdot b)$, die Laufzeit beträgt $O(m+n(k'+b))$. Sei $\gamma_{\text{ratio}} := \gamma_{\max}/\gamma_{\min}$: Mit der Wahl $b := \lceil \log(\gamma_{\text{ratio}})/\log\log(\gamma_{\text{ratio}}) \rceil$ ist analog zum Beweis des Satzes 5.6

$$k' \in \Theta(\log_b \gamma_{\text{ratio}}) = \Theta(\log \gamma_{\text{ratio}}/(\log\log \gamma_{\text{ratio}} - \log\log\log \gamma_{\text{ratio}})) = \Theta(b)$$

wodurch sich die behaupteten Schranken ergeben. □

Bei der Implementierung ist bzgl. des Lemmas 5.9 zu beachten, dass die Indizes der Level nicht bei 0, sondern bei $\lfloor \log_b \gamma_{\min} \rfloor$ beginnen.

Korollar 5.11 *Für $b = 2$ ist $k' = \log(\gamma_{\max}/\gamma_{\min})$ und es ergibt sich eine Laufzeit von $O(m+n\log(\gamma_{\max}/\gamma_{\min}))$ bei $O(\log(\gamma_{\max}/\gamma_{\min}))$ benötigtem Platz.*

Im Mittel erreicht man mit der Wahl $b = 2$ lineare Laufzeit. Dazu ist es jedoch nötig, im delete_min(·) das kleinste nichtleere Level in konstanter Zeit zu bestimmen. Dazu wird zu jedem der $k' := O(\log \gamma_{\max}/\gamma_{\min})$ Level ein Bit gespeichert, das angibt, ob $n_i = 0$ ist. Da dieses nur logarithmisch viele Bits sind, lassen sich diese in einer Speicherzelle kodieren. Es gilt ([AMT99]):

$$\forall i \leq k : i \text{ ist das höchstwertige gesetzte Bit } \Leftrightarrow (n_i \neq 0) \wedge \left(\bigwedge_{j=i+1}^{k} (n_j = 0) \right)$$

D.h., dass sich das höchstwertige Bit mit Hilfe einer AC^0-Operation (mit 2 Schichten) bestimmen lässt: Die erste Schicht gibt für jedes i abhängig von obiger Formel entweder 0 oder i aus, wobei bis auf ein i alle Ausgaben 0 sind. In der zweiten Schicht werden die Bits der einzelnen Ausgaben durch ein logisches Oder verbunden, sodass genau das gesuchte i als Ergebnis herauskommt. Da für die Praxis diese AC^0-Operation auf einem normalen Rechner nicht zur Verfügung steht, schlägt [Tho00] vor, alternativ die Zahl in eine Fließkommazahl zu wandeln und dann den Exponenten als Ergebnis zurückzugeben – diese Umwandlungen sind ebenfalls AC^0-Operationen und stehen auf heutigen Rechnern in der Regel zur Verfügung. Nach [Tho00] war in Experimenten diese Methode deutlich schneller, als mittels binärer Suche das höchstwertige Bit zu bestimmen.

Somit benötigen die Operationen insert(·), decrease_key(·) und delete_min(·) mit der Wahl $b = 2$ (oder einer anderen Konstanten) neben dem über die Knoten amortisierten Aufwand nur je $O(1)$ Schritte.

Für jeden Knoten ist über den Wert $\gamma_{\min}(v)$ die Anzahl der Level bestimmt, durch die der Knoten v höchstens wandern muss, bevor er in die Menge *Next*

aufgenommen wird. Wir werden nun zeigen, dass der Erwartungswert dieser Anzahlen summiert über alle Knoten durch $O(m)$ beschränkt ist, sodass der Multilevel-Bucket-Algorithmus unter Ausnutzung des Kaliberlemmas mit der Wahl $b = 2$ im Mittel nur $O(n + m)$ Schritte benötigt.

Das Kaliber $\gamma_{\min}(v) := \min\{\gamma(uv) \mid uv \in E\}$ eines Knotens v bestimmt die Anzahl der Level, durch die dieser Knoten höchstens wandern muss: Diese Anzahl beträgt nach Lemma 5.8 und 5.9 $k - \lfloor \log \gamma_{\min}(v) \rfloor$. D.h., ist $2^{i-1} \leq \gamma_{\min}(v) < 2^i$, so wandert der Knoten v durch höchstens $k - i + 1$ Level.[5] Ist $k := \lceil \log(\gamma_{\max}+1) \rceil$, so ist 2^k die kleinste 2er-Potenz, die echt größer als $\gamma_{\max}$ ist, und es ist $2^{k-1} \leq \gamma_{\max}$. Sind die Kantengewichte gleichverteilt in dem Intervall $[1, \ldots, \gamma_{\max}]$, so ist die Wahrscheinlichkeit, dass für eine Kante uv das Gewicht $2^{i-1} \leq \gamma(uv) < 2^i$ beträgt

$$\mathbf{Pr}[2^{i-1} \leq \gamma(uv) < 2^i] \leq 2^{i-1}/\gamma_{max} \leq 2^{i-1}/2^{k-1} = 2^{i-k}$$

Die Wahrscheinlichkeit, dass ein Knoten (aufgrund der einen eingehenden Kante uv) potentiell durch $j := k - i + 1$ Level wandert, ist also wegen $i - k = 1 - j$ durch 2^{1-j} beschränkt. Der Erwartungswert für die Anzahl der benutzten Level aufgrund der Kante uv für den Knoten v beträgt somit höchstens

$$\sum_{j=1}^{k} j \cdot 2^{1-j} \leq 2 \cdot \sum_{j=1}^{\infty} j \cdot 2^{-j} \leq 2 \cdot \sum_{j=1}^{\infty} \sum_{j'=j}^{\infty} 2^{-j} = 4$$

Hat v mehrere eingehende Kanten, so muss von diesen das minimale Kantengewicht bzw. die maximale Anzahl der durch diese Kante mit minimalem Gewicht implizierten zu durchwandernden Level betrachtet werden. Das Maximum der Werte ist jedoch kleiner als die Summe, und der Erwartungswert der Summe ist gleich der Summe der Erwartungswerte. Somit sind für alle Knoten zusammen höchstens $4m$ zu durchwandernde Level zu erwarten. Der Gesamtaufwand im Mittel ist damit linear. [Gol04] zeigt weiter, dass unter der Voraussetzung, dass die Kantengewichte voneinander unabhängig sind, die Linearzeit auch mit großer Wahrscheinlichkeit erreicht wird.

Neben diesem Average-Case-Linearzeit-Algorithmus gibt es noch weitere: Der chronologisch erste stammt von Meyer ([Mey01]), der Worst-Case beträgt bei diesem Verfahren jedoch $O(nm \log n)$. Eine spätere, verbesserte Variante ([Mey03]) benötigt $O(nm)$ und erreicht im Mittel ebenfalls Linearzeit. Kürzlich wurde von Hagerup ([Hag04]) noch eine weitere Variante vorgestellt,

[5] Ändert sich der Wert $\mathrm{D}(v)$ z.B. durch ein decrease_key$(\cdot)$ auf einen Wert $\mathrm{D}(v) \leq D_{\min} + \gamma_{\min}(v)$, so wird v direkt in die Menge *Next* eingefügt, ohne die Level bis $\lfloor \log_b \gamma_{\min}(v) \rfloor$ zu durchlaufen – daher kann die Anzahl der durchlaufenen Level auch kleiner sein.

die im Worst-Case $O(m + n \log n)$ Schritte benötigt (unter Verwendung von Fibonacci-Heaps). Nach Hagerups Darstellung sind die verwendeten Datenstrukturen einfacher, was aber nach unserer Einschätzung nicht zutrifft.[6] Für die Praxis ist der Algorithmus von [Gol04] sicherlich die bessere Wahl, da die Konstante im $O(\cdot)$ der Laufzeit deutlich geringer ist. Beide Algorithmen benutzen AC^0-Operationen, um über die höchstwertigen Bits eines Bitvektors schnell den nächsten zu bearbeitenden Knoten zu finden, sodass auch dies keinen praktischen Vorteil für die eine oder andere Variante auf heutigen Rechnern bringt.

Das Kaliberlemma ist implizit bereits im schon 1978 vorgestellten Lemma von [Din78] enthalten, welches man zur Auswahl eines Knotens v mit $\mathrm{D}(v) = \mathrm{d}(v)$ im Algorithmus 3.30 benutzen kann. Dabei werden auch hier die Knoten nicht mehr unbedingt in streng monoton wachsender Entfernung zum Startknoten besucht – ein ähnliches Lemma ist auch in [DF79] beschrieben.

Statt wie bei Dials Algorithmus (Abschnitt 5.1) jeden Knoten mit Wert $\mathrm{D}(\cdot) = i$ im Bucket $A[i]$ abzulegen, verwendet Dinitz Intervalle der Breite $\gamma_{\min}$ und legt die Knoten in den Bucket $A[\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor]$. Das Kaliberlemma 5.8 liefert einen einfachen Beweis, dass die so analog zu Dials Algorithmus aus dem ersten nichtleeren Bucket entnommenen Knoten korrekte Schätzentfernungen haben.

Lemma 5.12 ($\mathrm{D}(v) = \mathrm{d}(v)$ nach Dinitz) *Sei $v \in V - B$ mit $\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor = \min_{v' \in V-B} \{ \lfloor \mathrm{D}(v')/\gamma_{\min} \rfloor \}$, so gilt $\mathrm{D}(v) = \mathrm{d}(v)$.*

Beweis: Ist die Randmenge $R = \emptyset$, so ist $\mathrm{D}(v) = \infty,\ v \in V - B$, kein weiterer Knoten ist vom Startknoten aus erreichbar und das Lemma gilt direkt. Ist v ein Knoten aus dem ersten nichtleeren Bucket, so ist

[6] Etwas vereinfacht dargestellt ist die wesentliche Idee in Hagerups Algorithmus eine Aufteilung der Knoten in zwei Mengen: Abhängig von der Kantenanzahl wird ein Schwellwert bestimmt – Knoten, für die das Kaliber kleiner als dieser Schwellwert ist, befinden sich in der einen Menge (diese werden über Fibonacci-Heaps verwaltet), die restlichen Knoten in der anderen. Der Schwellwert ist so bestimmt, dass $O(n/\log n)$ Knoten in der einen Menge zu erwarten sind, diese benötigen dann im Mittel $O(m + (n/\log n) \cdot \log n) = O(m + n)$ Schritte. In der anderen Menge wird über die Bestimmung des höchstwertigen Bits eines Bitvektors der kleinste nichtleere Bucket gefunden – für die Knoten aus diesem gilt das Kaliberlemma, sodass für diese Knoten nur konstanter Aufwand nötig ist. Der zusätzliche Overhead summiert sich insgesamt ebenfalls zu $O(m)$, sodass der Erwartungswert der Laufzeit linear ist.

Hagerup schlägt auch eine Variante mit normalen Heaps vor. Die Konstante im $O(\cdot)$ dürfte hier entsprechend geringer sein, dafür ist der Worst-Case dann mit $O(m \log n)$ deutlich schlechter.

$\mathrm{D}_{\min} := \lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor \cdot \gamma_{\min} \leq \min(\mathrm{D}(R))$ eine untere Schranke für die Werte $\mathrm{D}(R)$. Es gilt mit $\lfloor (\mathrm{D}_{\min} + \gamma_{\min})/\gamma_{\min} \rfloor = \lfloor \mathrm{D}_{\min}/\gamma_{\min} \rfloor + 1 > \lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor$

$$\mathrm{D}_{\min} \leq \mathrm{D}(v) < \mathrm{D}_{\min} + \gamma_{\min} \leq \mathrm{D}_{\min} + \gamma_{\min}(v)$$

und mit dem Kaliberlemma 5.8 folgt $\mathrm{D}(v) = \mathrm{d}(v)$. □

Insbesondere ist für alle Knoten aus dem nichtleeren Bucket mit kleinstem Index die Entfernung $\mathrm{D}(v) = \mathrm{d}(v)$ korrekt, sodass die Knoten in beliebiger Reihenfolge entnommen werden können.

Satz 5.13 (Algorithmus nach [Din78]) *Die Laufzeit des Algorithmus 3.30 mit Lemma 5.12 im Graphen $G = (V, E, \gamma)$, $\gamma(\cdot) > 0$ mit Startknoten v_0 beträgt $O(n + m + \max_{v \in V}\{\mathrm{dist}(v_0, v)/\gamma_{\min}\})$ Schritte.*

Beweis: Die Menge $R := N(B) - B$ und die zugehörigen Werte $\mathrm{D}(R)$ werden analog zu Dials Algorithmus in einem Bucket-Array verwaltet. Jeder Knoten v befindet sich im Bucket mit Index $\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor$. Wird jeder Bucket als doppelt verkettete Liste verwaltet, so können das Einfügen und Löschen im Bucket und somit insert$(\cdot)$ und decrease_key$(\cdot)$ in konstanter Zeit $O(1)$ durchgeführt werden. Analog zum Korrolar 3.36 gilt, dass der Index des kleinsten nichtleeren Buckets monoton steigend ist, sodass der Aufwand des delete_min$(\cdot)$ summiert über alle Knoten $O(n + \max_{v \in V}\{\mathrm{dist}(v_0, v)/\gamma_{\min}\})$ beträgt und somit die obige Schranke für die Gesamtlaufzeit bewiesen ist. □

In Straßengraphen ist zu erwarten, dass die maximale Entfernung nur mit $O(\sqrt{n})$ wächst. Zwischen zwei Kreuzungen wird ein Minimalwert von ca. 10 Meter sicherlich nicht unterschritten, sodass für diese Graphenklasse für das Verfahren nach Dinitz ebenfalls Linearzeit zu erwarten ist.

Analog zu Dials Algorithmus lässt sich der Speicherbedarf senken, da nur ein kleiner Anteil der Buckets gleichzeitig belegt sein kann.

Lemma 5.14 *Der Algorithmus 3.30 mit Lemma 5.12 lässt sich mit Speicheraufwand $O(n + \gamma_{\max}/\gamma_{\min})$ implementieren.*

Beweis: Analog zu Lemma 5.2 folgt, dass

$$\begin{aligned}
&\max_{v \in R}\{\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor\} \\
&\quad \leq \left\lfloor \left(\max_{v \in R : \lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor = \min_{v' \in R}\{\lfloor \mathrm{D}(v')/\gamma_{\min} \rfloor\}} \{\mathrm{D}(v)\} + \gamma_{\max} \right) / \gamma_{\min} \right\rfloor \\
&\quad \leq \min_{v \in R}\{\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor\} + \lceil \gamma_{\max}/\gamma_{\min} \rceil
\end{aligned}$$

D.h., ist i der Index des ersten nicht leeren Buckets und j der Index des letzten nicht leeren Buckets, so ist $j-i$ im gesamten Verlauf des Algorithmus stets durch $\lceil \gamma_{\max}/\gamma_{\min} \rceil$ beschränkt. Es reicht also ein Bucketarray der Länge $\lceil \gamma_{\max}/\gamma_{\min} \rceil + 1$ aus und der Knoten v wird dann in den Bucket mit dem Index $\lfloor \mathrm{D}(v)/\gamma_{\min} \rfloor \mod (\lceil \gamma_{\max}/\gamma_{\min} \rceil + 1)$ abgelegt. □

5.5 Anwendung der Bucket-Strukturen bei reellen Kantengewichten

Ein in der Literatur noch nicht erwähnter Ansatz ist, über die Idee von [Din78] Bucket-Ansätze auch für positive reelle Kantengewichte zu verwenden.

Sind alle Kantengewichte $\gamma(\cdot) > 0$, so ist $\gamma_{\min} > 0$ und die Werte $\lfloor \mathrm{D}(\cdot)/\gamma_{\min} \rfloor$ sind ganzzahlig – dies lässt sich auch als Normierung auffassen, sodass $\gamma'_{\min} = 1$ und $\gamma'_{\max} = \gamma_{\max}/\gamma_{\min}$ sind. Werden die Knoten bzgl. dieser ganzzahligen Werte in monoton aufsteigender Reihenfolge aufgenommen, so folgt die Korrektheit direkt aus Lemma 5.12.

Die Anwendung dieser Idee auf Dials Algorithmus (Satz 5.1) ergibt genau den Algorithmus von Dinitz (Satz 5.13). Es folgt für die anderen im Kapitel 5 vorgestellten Algorithmen:

Korollar 5.15 *Die Überlaufbucket-Variante von Dials Algorithmus bzw. der 2-Level-Bucket-Algorithmus nach [DF79] und [CGR96] (Satz 5.4) lässt sich mittels des dinitzschen Lemmas 5.12 auf positive reelle Zahlen erweitern. Der Platzverbrauch beträgt dann* $O(\sqrt{\gamma_{\max}/\gamma_{\min}})$ *und die Laufzeit* $O(m + n \cdot \sqrt{\gamma_{\max}/\gamma_{\min}})$ *Schritte.*

Korollar 5.16 *Der Multilevel-Bucket-Algorithmus nach [DF79], [AMOT90], [CGR96], [CGS99], [Gol04] (Satz 5.6) lässt sich mittels des dinitzschen Lemmas 5.12 auf positive reelle Zahlen erweitern. Der Platzverbrauch beträgt dann* $O((\log(\gamma_{\max}/\gamma_{\min})/\log\log(\gamma_{\max}/\gamma_{\min}))^2)$ *und die Laufzeit* $O(m + n \cdot \log(\gamma_{\max}/\gamma_{\min})/\log\log(\gamma_{\max}/\gamma_{\min}))$ *Schritte.*

Korollar 5.17 *Der Multilevel-Bucket-Algorithmus mit Verbesserung nach [AMOT90] (Satz 5.7) lässt sich mittels des dinitzschen Lemmas 5.12 auf positive reelle Zahlen erweitern. Die Laufzeit beträgt dann* $O(m + n \cdot \sqrt{\log(\gamma_{\max}/\gamma_{\min})})$ *Schritte.*

Für gerichtete Graphen mit positiven reellwertigen Kantengewichten ist dies eine Verbesserung gegenüber dem Algorithmus nach [PR02] (Laufzeit $O(m + n \log(\gamma_{max}/\gamma_{\min})))$ und auch gegenüber Dijkstras Algorithmus mit Fibonacci-Heaps (Laufzeit $O(m + n \log n)$), sofern $\sqrt{\log(\gamma_{\max}/\gamma_{\min})} \in o(n)$. Die bisher schnellsten Algorithmen benötigen aber lediglich ein vergleichsbasiertes Modell. Die obige Anwendung der Bucketstrukturen setzt voraus, dass Divisionen und Rundung in konstanter Zeit möglich sind.

Bei Zufallsgraphen scheint es in der Praxis sinnvoll, für $\gamma_{\min}$ eine untere Schranke (z.B. 1) einzuführen. Dies behebt das Problem, dass das Verhältnis $\gamma_{\max}/\gamma_{\min}$ nicht vorab beschränkt ist, wenn die Zahlen gleichverteilt aus einem Intervall $(0 \dots c]$ genommen würden. Damit würden sich auch die Platzschranken vereinfachen.

Wendet man die obigen Ideen auf Algorithmen an, die die Knoten nicht mehr in aufsteigender Reihenfolge aufnehmen, muss ggf. auf Kriterien geachtet werden, die für die Korrektheit der entsprechenden Verfahren von Bedeutung sind.

Bei der Anwendung der Mutlilevel-Bucket-Variante mit Kaliberlemma nach [Gol04] auf Graphen mit reellwertigen Kantengewichten ergeben sich kaum Probleme. Im Multilevel-Bucket werden die Werte $\lfloor \mathrm{D}(\cdot)/\gamma_{\min} \rfloor$ abgespeichert. Wird über delete_min$(\cdot)$ ein Knoten aus der Bucketstruktur entnommen, so kann der exakte Wert $\mathrm{D}_{\min}$ gespeichert werden (statt $\lfloor \mathrm{D}_{\min}/\gamma_{\min} \rfloor$). Somit bringt die Anwendung des Kaliberlemmas beim Verteilen der Knoten eines Buckets oder bei einer insert$(\cdot)$-bzw. decrease_key$(\cdot)$-Operation keine Probleme mit sich. Bzgl. der erwarteten Linearzeit ergeben sich höchstens insofern Probleme, dass $\gamma_{\max}/\gamma_{\min}$ vorab nicht abgeschätzt werden kann – es stellt sich dann die Frage, inwiefern die Kantengewichte gleichverteilt sind. In der Abschätzung für die erwartete Laufzeit ist dieser Wert jedoch unerheblich. Letzten Endes spielt $\gamma_{\max}/\gamma_{\min}$ nur bzgl. des Aufwands für die Initialisierung der Datenstrukturen eine Rolle.

Kapitel 6

Skalierungstechniken

Im Kapitel 5 über Bucketstrukturen in Kürzeste-Wege-Algorithmen haben wir gesehen, dass die Laufzeit von der Größe des größten Kantengewichts $\gamma_{\max}$ abhängen kann. Mittels Skalierungstechniken kann man versuchen, das Problem der Kürzeste-Wege-Berechnung in einem Graphen zu vereinfachen, indem man das Problem auf einem reduzierten Graphen mit kleineren Kantengewichten löst und mit dem Ergebnis des reduzierten Problems eine Lösung für den Originalgraphen konstruiert.

Auf Graphen mit beliebigen ganzzahligen Kantengewichten ist der auf Skalierungstechniken basierende Algorithmus von Goldberg ([Gol95]) mit einer Laufzeit von $O(\sqrt{n}m\log(-\gamma_{\min}))$, $\gamma_{\min} \leq -2$ dem Verfahren von Bellman ([Bel58]) für $-\gamma_{\min} \in o(2^{\sqrt{n}})$ überlegen.

Auch bei den Skalierungsalgorithmen werden wieder Potentialfunktionen zum Einsatz kommen.

6.1 Der Skalierungsalgorithmus von Gabow

Zur Illustration der Idee der Skalierungstechnik soll hier zunächst der Algorithmus von Gabow ([Gab85]) vorgestellt werden, der auf Graphen mit ganzzahligen nichtnegativen Kantengewichten eine Laufzeit von $O((n+m)\cdot\log\gamma_{\max})$ erreicht.

Das Kernstück des Algorithmus basiert auf der Methode von Dial (Abschnitt 5.1), die in Graphen $G = (V, E, \gamma)$ mit $\gamma(\cdot) \in \{0, 1\}$ das SSSP-Problem nach Satz 5.1 in $O(m+n)$ löst. Im folgenden Algorithmus ist dieses SSSP-Problem als Teilproblem vielfach zu lösen.

Algorithmus 6.1 *SSSP-Skalierungsalgorithmus*

$A_{6.1}.1$ **if** $\gamma_{\max} \leq 1$ **then**
$A_{6.1}.1a.1$ $\mathrm{d}(\cdot) := \mathrm{dist}_G(v_0, \cdot)$ in $G = (V, E, \gamma)$
$A_{6.1}.1$ **else**
$A_{6.1}.1b.1$ $\mathrm{d}'(\cdot) := dist_{G'}(v_0, \cdot)$ in $G' = (V, E, \gamma')$ mit $\gamma'(\cdot) = \lfloor \gamma(\cdot)/2 \rfloor$;
$A_{6.1}.1b.2$ $\mathrm{d}^{(\pi)}(\cdot) := dist_{G_\pi}(v_0, \cdot)$ in $G_\pi := (V, E, \gamma_\pi)$ mit $\pi(\cdot) = 2 \cdot \mathrm{d}'(\cdot)$;
$A_{6.1}.1b.3$ $\mathrm{d}(\cdot) := 2 \cdot \mathrm{d}'(\cdot) + \mathrm{d}^{(\pi)}$
$A_{6.1}.1$ **fi**

Satz 6.2 *Der Algorithmus 6.1 löst (durch rekursive Anwendung) das SSSP-Problem in Graphen mit ganzzahligen nichtnegativen Kantengewichten in* $O((n+m) \cdot \log \gamma_{\max})$.

Beweis: Wir zeigen zunächst die Korrektheit des Algorithmus: Die kürzesten Entfernungen im Graphen $G' = (V, E, \gamma')$ mit $\gamma'(\cdot) = \lfloor \gamma(\cdot)/2 \rfloor$ sind eine Näherungslösung für $\mathrm{d}(\cdot)/2$ im Originalgraphen: Es gilt für alle $uv \in E$:

$$2 \cdot \gamma'(uv) \leq \gamma(uv) \leq 2 \cdot \gamma'(uv) + 1$$

Sei $w = v_0 v_1 \cdots v_k$ ein kürzester Weg in G, so hat w in G' die Länge

$$\gamma'(w) \leq \frac{1}{2} \cdot \mathrm{d}(v_k)$$

Die kürzeste Entfernung $\mathrm{d}'(v_k)$ kann höchstens noch kleiner als $\gamma'(w)$ sein. Sei $w = v_0 v_1 \cdots v_k$ ein kürzester Weg in G', so hat w in G die Länge

$$\gamma(w) \leq 2 \cdot \mathrm{d}'(v_k) + k \leq 2 \cdot \mathrm{d}'(v_k) + (n-1)$$

Die kürzeste Entfernung $\mathrm{d}(v_k)$ kann höchstens noch kleiner als $\gamma(w)$ sein. Somit gilt für alle $v \in V$

$$2 \cdot \mathrm{d}'(v) \leq \mathrm{d}(v) < 2 \cdot \mathrm{d}'(v) + n$$

D.h., in $G_\pi := (V, E, \gamma_\pi)$ mit $\pi(\cdot) = 2 \cdot \mathrm{d}'(\cdot)$, $\gamma_\pi(uv) = \pi(u) + \gamma(uv) - \pi(v)$ gilt mit $\pi(v_0) = 0$: $\mathrm{d}^{(\pi)}(v) = \mathrm{dist}(v_0, v) - \pi(v)$ und somit

$$0 \leq \mathrm{d}(v) - 2 \cdot \mathrm{d}'(v) = \mathrm{d}^{(\pi)}(v) = \mathrm{d}(v) - 2 \cdot \mathrm{d}'(v) < n$$

Alle kürzesten Entfernungen in G_π liegen zwischen 0 und $n-1$. Die Kantengewichte $\gamma_\pi(uv)$ sind aufgrund der Konsistenzeigenschaft kürzester Entfernungen ($\mathrm{d}'(v) \leq \mathrm{d}'(u) + \gamma'(uv)$) und obiger Eigenschaft $2 \cdot \gamma'(uv) \leq \gamma(uv)$

$$\gamma_\pi(uv) = 2 \cdot \mathrm{d}'(u) + \gamma(uv) - 2 \cdot \mathrm{d}'(v) \geq \gamma(uv) - 2 \cdot \gamma'(uv) \geq 0$$

Die Laufzeit von Dials Algorithmus ist $O(n+m+\max(d(V)))$, die kürzesten Entfernungen in G_π lassen sich somit stets in $O(n+m)$ bestimmen.[1]

Berechnet man die kürzesten Entfernungen in $G' = (V, E, \gamma')$ mit $\gamma'(\cdot) = \lfloor \gamma(\cdot)/2 \rfloor$ rekursiv mit Algorithmus 6.1, so gilt für die Laufzeit $T(\gamma_{\max}) = T(\gamma_{\max}/2) + O(n+m)$ und $T(1) = O(n+m)$, somit also $T(\gamma_{\max}) = O((n+m) \cdot \log \gamma_{\max})$. □

Möchte man auch den Kürzeste-Wege-Baum berechnen, so reicht es, dies in der letzten Iteration zu tun: Der Kürzeste-Wege-Baum in G_π ist mit dem in G nach Lemma 3.5 identisch.

Bemerkung 6.3 *Da für alle Kanten* $2 \cdot \gamma'(\cdot) \leq \gamma(\cdot) \leq 2 \cdot \gamma'(\cdot) + 1$ *gilt, folgt auch, dass der Kürzeste-Wege-Baum in* G' *zu einem Spannbaum in* G_π *wird, dessen Kanten alle Gewicht 0 oder 1 haben, sodass auch hieran anschaulich zu sehen ist, dass der längste Weg in* G_π *höchstens Länge* $n-1$ *hat.*

Bei Skalierung der Kantengewichte auf $\lfloor \gamma(\cdot)/(2+m/n) \rfloor$ bleibt die Länge $\max(d(V))$ in G_π durch $O(m)$ beschränkt, sodass pro Iteration der Aufwand weiterhin $O(n+m)$ beträgt – [Gab85] erreicht durch die geringere Anzahl der Iterationen die leicht verbesserte Laufzeit

$$O((n+m) \cdot \log_{(2+m/n)} \gamma_{\max}) = O((n+m) \cdot \log \gamma_{\max} / \log(2+m/n))$$

Bei Graphen mit $m \in \omega(n)$ ist dies bis zu einem Faktor $\log n$ schneller.

6.2 Der Skalierungsalgorithmus von Goldberg

Wendet man Skalierungstechniken auf Graphen mit negativen Kantengewichten an, muss beachtet werden, dass durch die Skalierung keine Zyklen negativer Länge eingeführt werden dürfen, da sonst in den Graphen mit den betragsmäßig kleineren Kantengewichten das SSSP-Problem unter Umständen

[1] Die durch die Potentialfunktion modifizierten Kantengewichte sind zwar alle positiv, können aber auch größer als 1 sein. Falls dadurch ein Wert $D(\cdot) \geq n$ auftreten sollte, kann dieser ignoriert werden, da die kürzesten Wege alle eine Länge kleiner n haben. Formal kann man die Knoten mit $D(\cdot) \geq n$ in einem Überlaufbucket wie in Abschnitt 5.2 verwalten – der Algorithmus terminiert, bevor Knoten aus dem Überlaufbucket betrachtet werden könnten.

nicht mehr lösbar ist. Dieses kann man erreichen, wenn bei der Halbierung ungerader Zahlen stets auf die nächstgrößere Zahl aufgerundet wird.[2] Dadurch kann es passieren, dass negative Zyklen beim Halbieren der Kantengewichte verschwinden, es können aber keine neuen negativen Zyklen hinzukommen. Beim Rückwärtsrechnen muss dann erkannt werden, ob der ursprüngliche Graph einen Zyklus negativer Länge hatte.

Das Berechnungsschema ist prinzipiell identisch: Wenn das kleinste Kantengewicht nichtnegativ ist, so löst man das SSSP-Problem z.B. mit Dijkstras Algorithmus in $O(m + n \log n)$. Ansonsten berechnet man in $G' = (V, E, \gamma')$ – nun mit $\gamma'(\cdot) = \lceil \gamma(\cdot)/2 \rceil$ – die kürzesten Entfernungen $\mathrm{d}'(\cdot)$, verwendet diese als Potentialfunktion $\pi(\cdot) = 2 \cdot \mathrm{d}'(\cdot)$ in G_π und erhält so die kürzesten Entfernungen $\mathrm{d}(\cdot) := 2 \cdot \mathrm{d}'(\cdot) + \mathrm{d}^{(\pi)}$.

Analog zu Gabows Algorithmus aus Abschnitt 6.1 gelten hier Beziehungen zwischen den Kantengewichten und kürzesten Entfernungen der beteiligten Graphen. Da bei der Skalierung auf- statt abgerundet wird, gilt nun für alle $uv \in E$:

$$2 \cdot \gamma'(uv) - 1 \leq \gamma(uv) \leq 2 \cdot \gamma'(uv)$$

Seien $w = v_0 \cdots v$ ein kürzester Weg in G und $w' = v_0 \cdots v$ einer in G', dann ist

$$\begin{aligned} 2 \cdot \mathrm{d}'(v) - (n-1) \leq 2 \cdot \gamma'(w) - (n-1) &\leq \gamma(w) \\ &= \mathrm{d}(v) \leq \gamma(w') \leq 2 \cdot \gamma'(w') = 2 \cdot \mathrm{d}'(v) \end{aligned}$$

Somit gilt für alle $v \in V$

$$2 \cdot \mathrm{d}'(v) - n < \mathrm{d}(v) \leq 2 \cdot \mathrm{d}'(v)$$

D.h., in $G_\pi := (V, E, \gamma_\pi)$ mit $\pi(\cdot) = 2 \cdot \mathrm{d}'(\cdot)$, $\gamma_\pi(uv) = \pi(u) + \gamma(uv) - \pi(v)$ gilt mit $\pi(v_0) = 0$: $\mathrm{d}^{(\pi)}(v) = \mathrm{dist}(v_0, v) - \pi(v)$ und somit

$$-n < \mathrm{d}^{(\pi)}(v) \leq 0$$

Alle kürzesten Entfernungen in G_π liegen zwischen $-(n-1)$ und 0. Insbesondere ist das kleinste Kantengewicht mindestens -1. Die kürzesten Entfernungen in G betragen wiederum $\mathrm{d}(\cdot) := 2 \cdot \mathrm{d}'(\cdot) + \mathrm{d}^{(\pi)}$. Hat G_π einen Zyklus negativer Länge, so auch G, und das SSSP-Problem ist nicht lösbar.

[2] Würde man weiterhin abrunden, so würden z.B. die Kantengewichte 1, 1, 1, -2 eines Zyklus positiver Länge zu 0, 0, 0, -1 abgerundet. Damit würde dieser Zyklus durch Halbieren der Kantengewichte zu einem negativen Zyklus – die kürzesten Wege wären dann nicht mehr wohldefiniert.

Analog zu Bemerkung 6.3 wird hier der Kürzeste-Wege-Baum in G' zu einem Spannbaum in G_π, dessen Kanten alle Gewicht -1 oder 0 haben, sodass wiederum anschaulich zu sehen ist, dass in G_π alle kürzesten Wege eine Länge zwischen $-(n-1)$ (da $\gamma_\pi(\cdot) \geq -1$) und 0 (aufgrund des Spannbaums) haben.

[Gol95] gibt (implizit) einen Algorithmus für das SSSP-Problem in Graphen mit ganzzahligen Kantengewichten $\gamma(\cdot) \geq -1$ (oder Finden eines negativen Zyklus) mit Laufzeit $O(\sqrt{n} \cdot m)$ an – mit diesem lassen sich die obigen Teilprobleme lösen. Die Gesamtlaufzeit zur Lösung des SSSP-Problems besteht dann aus logarithmisch häufigem Halbieren der Kantengewichte, Lösen eines SSSP-Problems mit nichtnegativen Kantengewichten und logarithmisch häufigem Lösen eines SSSP-Problems in Graphen mit $\gamma(\cdot) \geq -1$. Die Laufzeit ist demnach

$$O((n+m) \cdot \log(-\gamma_{\min}) + m + n \cdot \log n + \sqrt{n} \cdot m \cdot \log(-\gamma_{\min})) \\ = O(\sqrt{n} \cdot m \cdot \log(-\gamma_{\min}))$$

Betrachten wir $G_{\pi \leq 0}$, den Teilgraphen von G_π, der lediglich die Kanten mit $\gamma_\pi(\cdot) \leq 0$ enthält. Man könnte meinen, dass die kürzesten Wege von G_π alle in $G_{\pi \leq 0}$ liegen. Diese ließen sich dann wie folgt finden:

Man bestimme die starken Zusammenhangskomponenten in $G_{\pi \leq 0}$ (nach [CLRS01] in $O(n+m)$). Hat eine dieser Komponenten eine Kante mit $\gamma_\pi(\cdot) < 0$, so hat G_π und damit auch G einen negativen Zyklus und die kürzesten Wege sind in G nicht alle wohldefiniert. Sonst haben alle Kanten einer Komponente Länge $\gamma_\pi(\cdot) = 0$ und man ziehe die Komponenten zu je einem Knoten zusammen (Aufwand $O(n+m)$) – der resultierende Graph $G_{\pi \leq 0, sZ}$ ist azyklisch und man kann die kürzesten Wege mit dem Algorithmus aus Abschnitt 3.4.1 in $O(n+m)$ bestimmen. Die Knoten der zusammengezogenen Komponenten haben alle dieselbe Entfernung zum Startknoten.

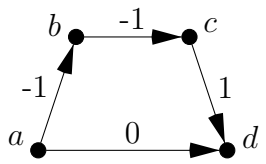


Abbildung 6.1: Fallstricke bei SSSP-Problemen mit $\gamma(\cdot) \geq -1$

Der Graph G in Abbildung 6.1 zeigt, dass die kürzesten Wege in G_π nicht notwendigerweise innerhalb $G_{\pi \leq 0}$ verlaufen müssen: Nach einmaliger Halbierung sind die Kantengewichte alle nichtnegativ. Die Berechnung der kürzesten Wege bestimmt alle $\mathrm{d}'(\cdot) = 0$ und G_π ist identisch mit dem Ausgangsgraphen G.

Die kürzesten Wege haben zwar wie oben gezeigt alle Länge kleiner gleich 0, der kürzeste Weg von a nach d beeinhaltet jedoch die Kante cd mit Gewicht 1.

Aufbauend auf folgender Beobachtung werden wir aber die kürzesten Wege in $G_{\pi \leq 0, sZ}$ zur Lösung des SSSP-Problems in G_π benutzen können: Wir identifizieren im Folgenden $\pi(\cdot)$ mit den Schätzentfernungen $\mathrm{D}(\cdot)$ in G.[3] Sei v ein Knoten mit einer eingehenden Kante uv mit $\gamma_\pi(uv) = -1$, d.h. uv ist eine verkürzende Kante. Verringert man $\pi(v)$ und $\pi(v')$ für alle von v aus erreichbaren Knoten v'[4] um jeweils 1, so ist danach $\gamma_\pi(uv) = 0$ und für alle anderen Kanten verringert sich $\pi(\cdot)$ entweder für den Anfangs- *und* Endknoten (d.h., $\gamma_\pi(\cdot)$ bleibt unverändert) oder nur für den Endknoten (d.h., $\gamma_\pi(\cdot)$ erhöht sich um 1). Insbesondere ist uv nicht mehr verkürzend und es wurden keine neuen verkürzenden Kanten erzeugt. Die Anzahl der Knoten, die eingehende Kanten mit $\gamma_\pi(\cdot) = -1$ haben, hat sich also verringert.

Diese Änderung der Potentialfunktion werden wir jetzt entweder entlang eines Pfades im Kürzeste-Wege-Baum von $G_{\pi \leq 0, sZ}$ benutzen oder für mehrere Knoten gleichzeitig durchführen, die in $G_{\pi \leq 0, sZ}$ gleich weit vom Startknoten entfernt sind.

Sei i die Anzahl der Knoten v, für die eine Kante uv mit $\gamma_\pi(uv) = -1$ im Kürzeste-Wege-Baum von $G_{\pi \leq 0, sZ}$ existiert. Sei weiterhin $i_{\mathrm{dist}} := \min(\mathrm{dist}_{G_{\pi \leq 0, sZ}}(v_0, \cdot))$ die kleinste Entfernung vom Startknoten. Da alle betrachteten Kanten Gewichte $\gamma_\pi(\cdot) \in \{-1, 0\}$ haben, gibt $|i_{\mathrm{dist}}|$ gleichzeitig die größte Anzahl von Kanten mit $\gamma_\pi(uv) = -1$ auf einem Weg in $G_{\pi \leq 0, sZ}$ an. Es können zwei Fälle auftreten:

1. $|i_{\mathrm{dist}}| \geq \sqrt{i}$: Sei $w = v_0 \cdots v_k$ der zugehörige Weg und i_1 bis $i_{|i_{\mathrm{dist}}|}$ die Indizes der Knoten mit $\gamma_\pi(\mathrm{pred}(v_{i_j})v_{i_j}) = -1$. Dann verringert man zunächst die Potentialfunktion $\pi(\cdot)$ für die von $v_{|i_{\mathrm{dist}}|}$ erreichbaren Knoten, zieht diese vorübergehend auf einen Knoten zusammen und wiederholt dies mit $v_{|i_{\mathrm{dist}}|-1}$ usw. bis v_{i_1}. Nun können die vorübergehend zusammengezogenen Knoten wieder entfaltet werden.

[3] Beginnen wir mit $\mathrm{D}(\cdot) := \pi(\cdot) = 2 \cdot \mathrm{d}'(\cdot)$, so sind wie oben gezeigt die bzgl. $\pi(\cdot)$ reduzierten Kantengewichte alle ≥ -1 und $\pi(\cdot)$ ist bzgl. den kürzesten Entfernungen $\mathrm{d}(\cdot)$ in G um höchstens $n - 1$ zu groß. Hat eine Kante Gewicht $\gamma_\pi(\cdot) < 0$, so ist sie im Sinne des generischen Algorithmus 3.30 verkürzend. Wir passen hier nun $\pi(\cdot)$ und somit auch gleichzeitig den Wert $\mathrm{D}(\cdot)$ an. Aufgrund der Gleichung 3.4 und des Lemmas 3.5 ist dies stets konsistent und konfliktfrei möglich. Haben am Ende alle bzgl. $\pi(\cdot)$ reduzierten Kanten Gewicht $\gamma_\pi(\cdot) \geq 0$, so gibt $\pi(\cdot) = \mathrm{D}(\cdot)$ nach Lemma 3.10 die korrekten Entfernungen in G an.

[4] Der Knoten u kann nicht erreichbar sein, da sonst ein negativer Zyklus $uv \rightsquigarrow u$ in $G_{\pi \leq 0, sZ}$ existieren würde – diese wurden bereits beim Bestimmen der starken Zusammenhangskomponenten erkannt.

Durch das vorübergehende Zusammenziehen müssen z.B. bei der Bearbeitung von $v_{|i_{\text{dist}}|-1}$ nicht die Werte $\pi(\cdot)$ aller von $v_{|i_{\text{dist}}|}$ erreichbaren Knoten (nochmals) reduziert werden, sondern nur der zusammengezogene. Dadurch wird jeder Knoten nur zweimal betrachtet: vor dem Zusammenziehen und beim Wiederentfalten. Dort muss dann der Wert $\pi(\cdot)$ jedes zusammengezogenen Knotens um 1 verringert werden. Ohne dieses vorübergehende Zusammenziehen wäre der Aufwand quadratisch statt linear.

Die Anzahl der Knoten, die eingehende Kanten mit $\gamma_\pi(\cdot) = -1$ haben, hat sich dadurch um mindestens $|i_{\text{dist}}| \geq \lceil\sqrt{i}\rceil$ verringert.

2. $|i_{\text{dist}}| < \sqrt{i}$: Dann gibt es keine Pfade mit vielen verkürzenden Kanten. Teilt man die Knoten mit eingehenden verkürzenden Kanten in die Mengen mit jeweils gleicher Entfernung zum Startknoten auf (dies können höchstens $\lceil\sqrt{i}\rceil-1$ Mengen sein), so muss eine dieser Mengen mehr als $\sqrt{i}$ Knoten beinhalten. Die Werte $\pi(\cdot)$ für die von diesen Knoten erreichbaren Knoten werden wieder um je 1 reduziert (ist ein Knoten von mehreren aus erreichbar, so darf nur einmal reduziert werden). Dadurch hat sich dann auch hier die Anzahl der Kanten mit $\gamma_\pi(\cdot) = -1$ um mindestens $i/|i_{\text{dist}}| \geq \lceil\sqrt{i}\rceil$ verringert.

Der Aufwand für jeden dieser Fälle ist linear. Es bleibt abzuschätzen, wieviel Iterationen nötig sind, um auf diese Weise die höchstens $n-1$ Knoten mit $\gamma_\pi(\cdot) = -1$ zu beseitigen – dieses werden $O(\sqrt{n})$ Iterationen sein.

Lemma 6.4 *Sei f eine Folge mit $f(0) = n$ und $f(i+1) = f(i) - \lceil\sqrt{f(i)}\rceil$. Ist $f(k-1) \neq 0$ und $f(k) = 0$, so ist $k \in O(\sqrt{n})$.*

Beweis: Solange $f(i) \geq n/2$ ist, reduziert sich $f(\cdot)$ in jedem Schritt um mindestens $\sqrt{n/2}$. D.h. $f(\sqrt{n/2}) \leq n - \sqrt{n/2} \cdot \sqrt{n/2} = n/2$. Nach weiteren $\sqrt{n/4}$ Iterationen ist $f(i)$ höchstens noch $n/4$. Ist $f(i) \leq 2$, so ist $f(i+1) = 0$. Es gilt somit

$$k \leq \sum_{j=1}^{\infty} \sqrt{\frac{n}{2^j}} = \frac{\frac{1}{\sqrt{2}}}{1-\frac{1}{\sqrt{2}}}\sqrt{n} = \frac{1}{\sqrt{2}-1}\sqrt{n} = \left(\sqrt{2}+1\right)\sqrt{n}$$

□

Der Graph $G_{\pi\leq 0,sZ}$ kann (und wird) sich nach jeder Iteration verändern. Es können neue Kanten mit Länge $\gamma_\pi(\cdot) = 0$ hinzukommen und sich somit die starken Zusammenhangskomponenten in $G_{\pi\leq 0,sZ}$ verändern. Da sich $\gamma_\pi(\cdot)$

immer nur um 1 verändert und – wie oben gezeigt – Kanten mit $\gamma_\pi(\cdot) = 0$ nicht verkürzend werden können, können auf diesem Wege keine Kanten mit $\gamma_\pi(\cdot) > 0$ verkürzend werden. Die Neuberechnung von $G_{\pi \leq 0,sZ}$ in jeder Iteration vergrößert nicht den asymptotischen Gesamtaufwand.

In $O(\sqrt{n})$ Iterationen mit jeweils $O(n + m)$ Aufwand ist also das SSSP-Problem in G mit $\gamma(\cdot) \geq -1$ zu lösen. Damit ist nun auch die Laufzeit von Goldbergs Skalierungsalgorithmus bewiesen.

Satz 6.5 *Das SSSP-Problem in Graphen mit $\gamma : E \to \mathbb{Z}$ lässt sich in $O(\sqrt{n} \cdot m \cdot \log(-\gamma_{\min}))$ Schritten lösen.*

Kapitel 7

Das APSP-Problem

Beim APSP-Problem werden die kürzesten Wege für alle Knotenpaare gesucht. Die (im asymptotischen Sinne) schnellsten Algorithmen zur Lösung des APSP-Problems sind in der Tabelle 7.1 aufgeführt.

$G = (V, E)$, $\gamma : E \to \cdot$	Laufzeit
G gerichtet, $\gamma : E \to \mathbb{R}$	[Joh77],[Dij59],[FT87]: $O(nm + n^2 \log n)$ [Fre76],[Tak92]: $O(n^3\sqrt{(\log\log n)/(\log n)})$ [Pet02], [Pet04]: $O(nm + n^2 \log\log n)$
G gerichtet, $\gamma : E \to \mathbb{Z}$	[Zwi02]: $\tilde{O}(\gamma_{\lvert\max\rvert}^{0.616} \cdot n^{2.616})$
G unger., $\gamma : E \to \mathbb{R}$	[PR02]: $O(nm\alpha(m, n))$
G unger., $\gamma : E \to \mathbb{N}$	[Tho99]: $O(nm)$ [SZ99]: $\tilde{O}(\gamma_{\max} \cdot n^{2.376})$

Tabelle 7.1: Übersicht zu APSP-Algorithmen

7.1 Einfache APSP-Algorithmen

Ein sehr bekannter APSP-Algorithmus geht auf [Flo62] zurück (siehe auch z.B. [CLRS01]) und löst in $O(n^3)$ Schritten das APSP-Problem. Nach der Initialsierung der $\mathrm{D}_u(v) := \gamma(uv)$ für $uv \in E$ ($\mathrm{D}_u(v) := \infty$ sonst, sowie $\mathrm{D}_v(v) := 0$) wird für alle Knoten v_k geprüft, ob der Weg von v_i nach v_j über v_k die bisher angenommene obere Schranke $\mathrm{D}_{v_i}(v_j)$ nach unten korrigiert. Nach dem Schleifendurchlauf für v_k sind die $\mathrm{D}_{v_i}(v_j)$ korrekt, wenn als Zwischenknoten für die Wege nur die bisher betrachteten v_k erlaubt wären.

F.1 **for all** $v_k \in V$ **do**
F.1.1 **for all** $v_i, v_j \in V$ **do** $\mathrm{D}_{v_i}(v_j) := \min\{\mathrm{D}_{v_i}(v_j), \mathrm{D}_{v_i}(v_k) + \mathrm{D}_{v_k}(v_j)\}$ **od**
F.1 **od**

Für dichte Graphen, d.h. $m \in \Theta(n^2)$, ist dieser einfache Algorithmus bereits sehr gut, nur die Matrixmultiplikationsalgorithmen sind dann asymptotisch schneller. Hat der Graph negative Zyklen, so kann dies an negativen Werten $\mathrm{D}_v(v)$ erkannt werden.

Ein wichtiges Hilfsmittel zur Beschleunigung bieten auch beim APSP-Problem die Potentialfunktionen. Mit dem Aufwand zur Lösung eines SSSP-Problems kann das APSP-Problem für beliebige Graphen nach Lemma 3.14 auf die n-fache Berechnung eines SSSP-Problems auf Graphen mit nichtnegativen Kantengewichten reduziert werden.

Satz 7.1 *Das APSP-Problem lässt sich in einem vergleichsbasierten Modell in $O(nm + n^2 \log n)$ lösen.*

Beweis: Man berechne mit Hilfe des Bellman-Ford-Algorithmus (Abschnitt 3.3.1) in $O(nm)$ die kürzesten Entfernungen $\mathrm{d}(\cdot)$ in $G' = (V \cup \{v_0\}, E \cup (\{v_0\} \times V), \gamma')$, $v_0 \notin V$ mit $\gamma'(uv) := \gamma(uv),\ uv \in E$, $\gamma'(v_0 v) := 0,\ v \in V$ von v_0 zu allen anderen Knoten und verwende dann $\pi(v) := \mathrm{d}(v)$ als Potentialfunktion. Nach Lemma 3.14 gilt in G_π für die reduzierten Kantengewichte $\gamma_\pi(\cdot) \geq 0$, sodass für jeden Startknoten mit Hilfe des Dijkstra-Algorithmus das SSSP-Problem in $O(m + n \log n)$ gelöst werden kann (Abschnitt 3.4.3). Die in G_π berechneten Entfernungen lassen sich mit Gleichung 3.4 in $O(n^2)$ in die kürzesten Entfernungen in G übertragen. Die Kürzeste-Wege-Bäume in G_π und G sind nach Lemma 3.5 identisch. □

7.2 Ein APSP-Algorithmus für Graphen mit negativen Zyklen

Hat ein Graph G negative Zyklen, so kann es wegen Lemma 3.3 keine Potentialfunktion π mit $\gamma'(uv) = \gamma(uv) + \pi(u) - \pi(v) \geq 0$ für alle $uv \in E$ geben. Ebenso werden nicht für alle Knotenpaare kürzeste Wege wohldefiniert sein – liegt auf einem Weg von u nach v ein negativer Zyklus, so gibt es zu jedem Weg von u nach v einen kürzeren Weg, indem der negative Zyklus ausreichend oft durchlaufen wird. Erlaubt man für die kürzesten Entfernungen auch die Werte $\mathrm{dist}(\cdot,\cdot) = \infty$ (falls kein Weg existiert) und $\mathrm{dist}(\cdot,\cdot) = -\infty$ (falls ein

Weg mit negativem Zyklus existiert), so lassen sich für alle Knotenpaare die in diesem Sinne kürzesten Entfernungen ausgeben.

Satz 7.2 *Das APSP-Problem lässt sich in Graphen mit negativen Zyklen im obigen Sinne in $O(nm + n^2 \log n)$ lösen.*

Beweis: [MPSS02] berechnen dazu zunächst die starken Zusammenhangskomponenten und den Graphen G_{sZ}, der entsteht, wenn diese zu je einem Knoten zusammengezogen werden. Letzterer ist azyklisch ([CLRS01]), der Aufwand beträgt $O(n + m)$ Schritte incl. topologischer Sortierung von G_{sZ}. Innerhalb der starken Zusammenhangskomponenten kann mit je einer SSSP-Berechnung nach Satz 3.24 (Bellman-Ford-Algorithmus) in $O(\sum_i n_i m_i) \subseteq O(\sum_i n_i m) = O(nm)$ festgestellt werden, ob jeweils innerhalb einer Zusammenhangskomponente ein negativer Zyklus existiert – Zyklen können nur innerhalb einer starken Zusammenhangskomponente verlaufen.

In dem Graphen $G_{sZ} = (K, E_K)$ kann mit dieser Vorabinformation nun für jede Komponente $k \in K$ in $O(|K| + |E_K|) \subseteq O(n + m)$ (Algorithmus für azyklische Graphen, Abschnitt 3.4.1) bestimmt werden, zu welchen Komponenten im Ausgangsgraphen G keine Wege, Wege mit negativen Zyklen oder kürzeste Wege (mit endlichem Wert) bestehen. Dazu wird auf der eingeschränkten Wertemenge $\{\boxminus, \boxdot, \boxplus\}$ mit der Ordnung $\boxminus < \boxdot < \boxplus$ gearbeitet. Von Komponenten mit negativen Zyklen ausgehende Kanten haben Gewicht $\gamma(\cdot) = \boxminus$, sonst ist $\gamma(\cdot) = \boxdot$. Die Rechenregeln sind $\boxminus + \boxminus = \boxminus$ sowie $\boxminus + \boxdot = \boxminus$ und $\boxdot + \boxdot = \boxdot$. Die von Knoten mit $\mathrm{D}(\cdot) = \boxplus$ ausgehende Kanten werden ignoriert, daher können andere Summen nicht vorkommen. Initialisiert wird jeweils mit $\mathrm{D}(\cdot) := \boxplus$ und $\mathrm{D}(k) := \boxminus$ oder $\mathrm{D}(k) := \boxdot$, abhängig davon, ob die (Start-)Komponente k einen negativen Zyklus enthält oder nicht. So sind nach $O(n(n + m))$ Schritten alle Knotenpaare bestimmt, für die $\mathrm{dist}(\cdot, \cdot) = +/ - \infty$ gilt.

Es verbleibt die Berechnung der kürzesten Wege mit endlichem Wert. Dazu können die Knoten aus den Komponenten mit negativen Zyklen und die inzidenten Kanten in G entfernt werden, da für diese $\mathrm{dist}(\cdot, \cdot) = +/ - \infty$ gelten muss. In dem resultierenden Graphen kann das APSP-Problem wie in Satz 7.1 in $O(nm + n^2 \log n)$ gelöst werden. Die kürzesten Entfernungen mit endlichem Wert sind identisch zu denen in G. Nur falls auf dem Weg vorher eine Komponente mit negativem Zyklus lag, hat sich die kürzeste Entfernung vergrößert (und war oben mit $-\infty$ bestimmt). In $O(n^2)$ lassen sich dann die korrekten Werte $\mathrm{dist}(\cdot, \cdot)$ ausgeben. □

7.3 Der essentielle Teilgraph

Existiert zu einer Kante uv ein alternativer Weg $w = u \cdots v$ mit $\gamma(w) \leq \gamma(uv)$, so kann man die Kante uv aus dem Graphen streichen, ohne dass sich dadurch die kürzesten Entfernungen ändern.

Lemma 7.3 *Sei $G = (V, E, \gamma)$ mit $\gamma(\cdot) > 0$, so existiert ein Graph $H = (V, E', \gamma')$, $E' \subseteq E$ mit $\gamma'(\cdot) = \gamma(\cdot)$, $uv \in E'$, sodass sich die kürzesten Entfernungen* $\mathrm{dist}(u, v)$, *$u, v \in V$ in G und H nicht unterscheiden. Ist $|E'|$ minimal, so ist H eindeutig. Der Graph H mit $|E'|$ minimal heißt der* essentielle Teilgraph *von G.*

Beweis: Da die Kantengewichte echt positiv sind, können die Alternativwege $w = u \cdots v$ zur Kante uv mit $\gamma(w) \leq \gamma(uv)$ nur aus Kanten $u'v'$ mit $\gamma(u'v') < \gamma(uv)$ bestehen. Sortiert man die Kanten $m_i = u_i v_i$ in G aufsteigend nach Gewicht, so lässt sich H schrittweise eindeutig konstruieren:

1 $E' = \emptyset$;
2 **for** $i := 1$ **to** m **do**
2.1 **if** $\gamma(u_i v_i) < \mathrm{dist}_{H:=(V,E')}(u_i, v_i)$ **then** $E' := E' \cup \{u_i v_i\}$ **fi od**

Eine Kante $u_i v_i$ wird nur dann in E' aufgenommen, wenn es keinen höchstens gleich langen Weg mit kürzeren Kanten aus E gibt.

Da H Teilgraph von G ist, gilt $\mathrm{dist}_G(\cdot, \cdot) \leq \mathrm{dist}_H(\cdot, \cdot)$. Sei $w = v_0 \cdots v_k$ ein kürzester Weg in G. Ist die Kante $v_i v_{i+1} \notin E'$, so existiert nach Konstruktion ein (höchstens) gleich langer Weg $v_i \cdots v_{i+1}$, der statt der Kante $v_i v_{i+1}$ in w benutzt werden kann. Der so konstruierte Weg w' in H hat somit Länge $\gamma(w') \leq \gamma(w)$. Die kürzesten Entfernungen in G und H unterscheiden sich also nicht. □

Sind Kantengewichte $\gamma(\cdot) = 0$ erlaubt, so gilt die Aussage nicht mehr, da z.B. im vollständigen Graphen K mit $\gamma \equiv 0$ jeder Zyklus C, der alle Knoten enthält, die gleichen Entfernungen zwischen allen Knotenpaaren hat wie der Graph K. Somit ist die Menge der Kanten nicht eindeutig. Der obige Algorithmus garantiert dann nicht einmal, dass das berechnete E' minimal ist.[1]

[1] Dazu müssen nicht alle Kantengewichte gleich null sein. Z.B. in $G = (\{v_1, v_2, v_3\}, \{v_1v_2, v_1v_3, v_2v_3\}, \gamma)$ mit $\gamma(v_1v_2) = 0$ und $\gamma(v_1v_3) = \gamma(v_2v_3) = 1$ hängt es davon ab, ob v_1v_3 in der Sortierung vor v_2v_3 steht, damit v_1v_3 in E' mit aufgenommen wird oder nicht.

Satz 7.4 (APSP-Algorithmus nach [McG95]) *Sei $G = (V, E, \gamma)$, $\gamma(\cdot) > 0$, so lässt sich $H = (V, E', \gamma)$, der essentielle Teilgraph von G, in $O(n \cdot (|E'| + n \log n))$ berechnen und das APSP-Problem in G ebenfalls in $O(n \cdot (|E'| + n \log n))$ lösen.*

Beweis: [McG95] berechnen mit dem Algorithmus aus Lemma 7.3 schrittweise den essentiellen Teilgraphen H. Mit dem jeweils schon bestimmten Teil von E' werden parallel n SSSP-Probleme gelöst. Nach Lemma 7.3 stimmen die kürzesten Entfernungen in G und H überein. Die Laufzeit beträgt $O(n \cdot (|E'| + n \log n))$ zur Lösung der SSSP-Probleme, dies dominiert die Laufzeit $O(m \log m) \subseteq O(n^2 \log n)$ zur aufsteigenden Sortierung der m Kanten.

Da die Berechnung von H und die Lösung der SSSP-Probleme miteinander verwoben ist, bleibt zu zeigen, wie der Dijkstra-Algorithmus modifiziert werden muss und dass diese Modifizierung die asymptotische Laufzeit nicht erhöht.

Bei jeder Abfrage $\gamma(u_i v_i) < \text{dist}_{H:=(V,E')}(u_i, v_i)$, die Kante $u_i v_i$ ist (noch) nicht in E', wird das SSSP-Problem mit Startknoten u_i weiter bearbeitet, falls nicht bereits $\text{D}_{u_i}(v_i) \leq \gamma(u_i v_i)$ bekannt ist – in diesem Fall wird $u_i v_i$ verworfen und nicht in E' aufgenommen. Seit dem letzten Bearbeiten dieses SSSP-Problems sind einige Kanten $u'v'$ neu in E' aufgenommen worden, diese werden zunächst bearbeitet (durch insert$(\cdot)$ bzw. decrease_key$(\cdot)$ mit Fibonacci-Heaps amortisiert in je $O(1)$). Da die Kantenlängen dieser noch nicht berücksichtigten Kanten größer als der bisher größte Wert eines Knotens in diesem Kürzeste-Wege-Baum sind, kann durch das nachträgliche Bearbeiten kein Weg zu einem Baumknoten verkürzt werden – der Kürzeste-Wege-Baum bleibt unverändert. Nun wird der Dijkstra-Algorithmus weitergeführt bis entweder der Fibonacci-Heap leer ist oder ein Knoten mit $\text{D}_{u_i}(\cdot) > \gamma(u_i v_i)$ aufgenommen würde (dann gehört $u_i v_i$ zu E' dazu). Wenn v_i mit $\text{D}_{u_i}(v_i) \leq \gamma(u_i v_i)$ zum Baumknoten wurde, existiert in H bereits ein anderer nicht längerer Alternativweg und die Kante $u_i v_i$ wird verworfen. Der Ablauf verändert sich gegenüber dem normalen Dijkstra-Algorithmus nur insofern, dass Kanten mit größerem Gewicht nachträglich bearbeitet werden. Für jede Kante in E' ist dies ein Zusatzaufwand von $O(1)$, sodass die Laufzeit für jedes der SSSP-Probleme durch $O(|E'| + n \log n)$ beschränkt bleibt. □

Erlaubt man auch Kantengewichte $\gamma(\cdot) = 0$, so erhöht sich die Kardinalität von E' auf höchstens $|\{uv \mid \gamma(uv) = \text{dist}(u, v)\}|$. In dem oben genannten Beispiel des vollständigen Graphen wären dies $|E'| = n \cdot (n-1)/2$, während n Kanten in dem Teilgraphen ausreichen würden. In der Praxis wird der Unterschied kaum spürbar sein, da es unwahrscheinlich ist, dass zu einer

Kante uv mit $\gamma(uv) = \text{dist}(u, v)$ ein Alternativweg w mit $\gamma(w) = \text{dist}(u, v)$ existiert.

[KKP93] geben einen weiteren Algorithmus mit gleicher asymptotischer Laufzeit an, der dabei aber nicht ohne Weiteres die Kürzeste-Wege-Bäume gleichzeitig mitberechnen kann.

Für vollständig verbundene Zufallsgraphen (Kantengewichte gleichverteilt aus dem Intervall $[0, 1]$) geben [HZ85] an, dass mit hoher Wahrscheinlichkeit nur $O(n \log n)$ Kanten tatsächlich zu den kürzesten Wegen gehören. D.h., dass in solchen Zufallsgraphen die Algorithmen von [KKP93] und [McG95] das APSP-Problem mit hoher Wahrscheinlichkeit in nur $O(n^2 \log n)$ Schritten lösen. Für eine andere Klasse von Zufallsgraphen geben [MT87] ebenfalls einen Algorithmus mit mittlerer Laufzeit $O(n^2 \log n)$ an.

Eine Anwendung auf Graphen mit negativen Kantengewichten ist hier nicht ohne Weiteres möglich. Der Trick über Potentialfunktionen wie in Satz 7.1 greift hier nicht, da der Bellman-Ford Algorithmus bereits $O(nm)$ Schritte benötigt und damit den Vorteil durch den essentiellen Teilgraphen auffrisst. Sind die Kantengewichte ganzzahlig, kann jedoch Goldbergs Skalierungsalgorithmus aus Abschnitt 6.2 verwendet werden, der Worst-Case beträgt dann $O(\sqrt{n} \cdot m \cdot \log(-\gamma_{\min}) + n \cdot (|E'| + n \log n))$ Schritte.

7.4 Matrixmultiplikation und das APSP-Problem

Ist der Graph als Adjazenzmatrix $A = (a_{ij}) = (\gamma(v_i v_j)) \in (\mathbb{R} \cup \infty)^{n \times n}$, $a_{ii} = 0$ gegeben, so lassen sich mit der über Minimumbildung und Addition gebildeten Matrixmultiplikation die kürzesten Wege berechnen. Die Matrix $A \cdot A = A^2 = (a'_{ij})$ mit $a'_{ij} = \min_k\{a_{ik} + a_{kj}\}$ gibt die kürzesten Entfernungen mit höchstens zwei Kanten an, $(A^2)^2$ somit die kürzesten Entfernungen mit vier Kanten. Da ein kürzester Weg höchstens $n - 1$ Kanten haben kann, reichen logarithmisch viele Quadrierungen der Adjazenzmatrix aus, um die kürzesten Entfernungen zu bestimmen (negative Werte in der Diagonalen zeigen negative Zyklen an). Der Aufwand beträgt $O(n^3 \log n)$ – dabei wird jedoch nur das APD-Problem gelöst. Mit Hilfe des Lemmas 3.7 lassen sich ohne asymptotischen Mehraufwand auch die kürzesten Wege rekonstruieren.

Bei der normalen Matrixmultiplikation lässt sich nach der Methode von Strassen mittels Divide-and-Conquer der Aufwand auf $O(n^{\log 7}) = O(n^{2.8074})$ senken ([Str69]). [CW90] senken die obere Schranke für den Exponenten bei der Matrixmultiplikation weiter auf $\mathcal{M} := 2.376$. Leider können diese Ansätze

nicht direkt übertragen werden, da dazu eine Inverse der Minimumbildung gebraucht würde. Nichtsdestotrotz gibt es verschiedene Ansätze, Matrixmultiplikation bei der Lösung von APSP-Problemen einzusetzen.

[Fre76] zeigt, dass zur Berechnung des APSP-Problems $O(n^{5/2})$ Vergleiche und Additionen ausreichen. Die Algorithmen sind jedoch direkt von dem n abhängig, sodass für jedes n ein eigener Algorithmus nötig ist. Mit nur einem Algorithmus lässt sich sein Ansatz nach [Tak92] in ein $O(n^3 \cdot \sqrt{\log\log n / \log n})$-Verfahren umsetzen.

Für ungerichtete Graphen mit positiven ganzzahligen Kantengewichten geben [SZ99] einen $\tilde{O}(\gamma_{\max} \cdot n^{\mathcal{M}})$-Algorithmus an. Für gerichtete Graphen gibt [Zwi02] einen Algorithmus mit Laufzeit $O(n^{2.575})$ für Kantengewichte $\gamma(\cdot) \in \{-1, 0, 1\}$ an, sowie einen Algorithmus mit Laufzeit $\tilde{O}(\gamma_{|\max|}^{0.616} \cdot n^{2.616})$, falls auch betragsmäßig größere ganze Zahlen erlaubt sein sollen. In [Zwi01] findet man eine recht aktuelle Übersicht über weitere Algorithmen zu APSP-Algorithmen, speziell solche, die auf Matrixmultiplikation basieren.

Kapitel 8

Berechnung der k-kürzesten Wege

1959, fast zeitgleich mit der Veröffentlichung der ersten Kürzeste-Wege-Algorithmen, diskutieren Hoffman und Pavley bereits in [HP59] Algorithmen für eine Verallgemeinerung der Kürzeste-Wege-Suche, nicht nur den kürzesten, sondern auch den zweit-, dritt- oder allgemein k-kürzesten Weg zu suchen. Dabei unterscheidet man im Wesentlichen zwei Problemklassen: das uneingeschränkte und das eingeschränkte k-kürzeste-Wege-Problem. Beim letzteren werden nur zyklenfreie Wege berechnet. Neben der Einschränkung „nur zyklenfreie Wege“ sind noch andere Nebenbedingungen denkbar, auf diese wird hier aber nicht weiter eingegangen.

Beim k-kürzeste-Wege-Problem können Schlingen (Kanten, die von einem Knoten zu demselben Knoten zurückführen) und Multikanten (mehrere Kanten mit denselben Anfangs- und Endknoten) eine Rolle spielen, während bei kürzesten Wegen Schlingen nicht vorkommen können und bei Multikanten nur die Kante mit kleinstem Gewicht von Bedeutung ist. In der Literatur wie auch hier werden trotzdem Multikanten verboten, diese lassen sich bei Bedarf leicht durch Einfügen eines Knotens und Zweiteilung der parallel verlaufenden Kante simulieren. Wie bei kürzesten Wegen setzen wir auch hier grundsätzlich voraus, dass die Graphen keine negativen Zyklen haben.

Wir zeigen einige grundlegende Eigenschaften k-kürzester Wege, skizzieren den Aufbau der Algorithmen und geben eine neue einfache Implementierung für das uneingeschränkte k-kürzeste-Wege-Problem an, deren Laufzeit $O(m+n\log n+k(n+\log k))$ ist. Der Algorithmus verwendet nur sehr einfache Datenstrukturen (Heaps) und ist bis auf den deutlich komplizierteren Algorithmus von Eppstein ([Epp98]) der schnellste bekannte Algorithmus. Der

bisher schnellste Algorithmus nach [Fox73] – verbessert mit Fibonacci-Heaps – hat eine Laufzeit von $O(m + k(n \log n + \log k))$.

Eine ausführliche Literaturübersicht gibt David Eppstein in [Epp01], einem Verzeichnis von knapp 500 Veröffentlichungen auf dem Gebiet k-kürzester Wege. Die Ansätze lassen sich im Wesentlich in drei Kategorien einteilen: Algorithmen, die durch Verdoppeln von Wegen und Löschen von Kanten schon berechnete kürzere Wege vermeiden (z.B. Azevedo in [ACMM93], [AMMP94]), Algorithmen, die Methoden zur Lösung linearer Gleichungssysteme auf die Wegeberechnung übertragen (z.B. [Shi76]), und schließlich Algorithmen, die Eigenschaften analog zur Teilwegeoptimalität bei kürzesten Wegen ausnutzen. Der letztere Ansatz hat sich in Theorie und Praxis durchgesetzt. Wir werden die wesentlichen Konzepte hier vorstellen und orientieren uns dabei grob an [MPS99].

8.1 Das uneingeschränkte k-kürzeste-Wege-Problem

Analog zum Lemma 3.1 gilt für k-kürzeste Wege eine Optimalitätseigenschaft.

Lemma 8.1 *Seien $G = (V, E, \gamma)$ ein gewichteter gerichteter Graph und $w = v_0 v_1 \cdots v_z$ ein k-kürzester Weg von v_0 nach v_z, so gilt: $\forall i, j \in \{0, \ldots, z\}$, $i < j$ ist die Länge des Weges $w_{ij} := v_i v_{i+1} \cdots v_j$ nicht größer als die Länge des k-kürzesten Weges von v_i nach v_j.*[1]

Beweis: Angenommen, es gibt einen Weg $w_{ij} = v_i \cdots v_j$, der echt länger als der k-kürzeste Weg von v_i nach v_j ist, so existieren k verschiedene echt kürzere Wege $w'_{ij} = v_i \cdots v_j$ mit $\gamma(w'_{ij}) < \gamma(w_{ij})$ und somit auch k verschiedene Wege $w' = v_0 \overset{w_{0i}}{\rightsquigarrow} v_i \overset{w'_{ij}}{\rightsquigarrow} v_j \overset{w_{jz}}{\rightsquigarrow} v_z$ mit $\gamma(w') = \gamma(w) - \gamma(w_{ij}) + \gamma(w'_{ij}) < \gamma(w)$ im Widerspruch entweder zur Annahme, w_{ij} sei länger als der k-kürzeste Weg von v_i nach v_j, oder zur Annahme, w sei ein k-kürzester Weg von v_0 nach v_z. □

Verbietet man Zyklen, so stimmt diese Aussage nicht mehr. Im Beispiel in Abbildung 8.1 gibt es von v_0 nach v_1 genau zwei zyklenfreie Wege:

[1] Sind mehrere Wege von v_i nach v_j gleich lang, so kann w_{ij} k'-kürzester Weg von v_i nach v_j mit $k' > k$ sein. Ordnet man gleich lange Wege lexikografisch, so ist w_{ij} stets k'-kürzester Weg mit $k' \leq k$.

$w_1 = v_0v_1$ mit $\gamma(w_1) = 0$ und $w_2 = v_0v_3v_4v_1$ mit $\gamma(w_2) = 5$. Von v_0 nach v_3 gibt es drei zyklenfreie Wege: $w_3 = v_0v_1v_2v_3$ mit $\gamma(w_3) = 0$, $w_4 = v_0v_1v_3$ mit $\gamma(w_4) = 1$ und $w_5 = v_0v_3$ mit $\gamma(w_5) = 2$. Der drittkürzeste Weg w_5 von v_0 nach v_3 ist somit ein Teilweg des zweitkürzesten Weges w_2 von v_0 nach v_1. Lässt man Zyklen zu, so gilt Lemma 8.1 wie oben bewiesen wurde.

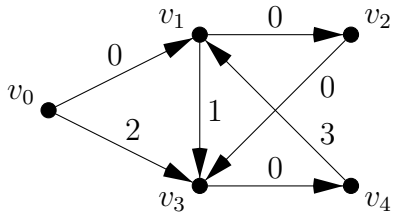


Abbildung 8.1: Gegenbeispiel zum Lemma 8.1 im eingeschränkten k-kürzeste-Wege-Problem

Die k kürzesten Wege lassen sich über die gemeinsamen Präfixe in einem Baum anordnen. Die Knoten der Wege werden zu den Knoten im Baum. Der kürzeste Weg lässt den Baum zunächst als Liste erscheinen. Fügt man den zweitkürzesten Weg hinzu, so wird dieser einen Teil des kürzesten Weges benutzen und dann von diesem abzweigen. Analog dazu kann man die nächstkürzesten Wege einfügen. In Abbildung 8.2 ist beispielhaft der 4-kürzeste-Wege-Baum für die 4 kürzesten Wege von v_0 nach v_1 dargestellt, die Wege sind $w_1 = v_0v_1$ mit $\gamma(w_1) = 0$, $w_2 = v_0v_1v_2v_3v_4v_1$ mit $\gamma(w_2) = 3$, $w_3 = v_0v_1v_3v_4v_1$ mit $\gamma(w_3) = 4$ und $w_4 = v_0v_3v_4v_1$ mit $\gamma(w_4) = 5$.

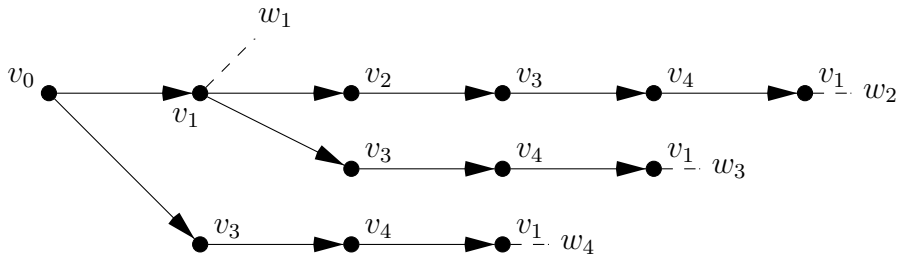


Abbildung 8.2: 4-kürzeste-Wege-Baum von v_0 nach v_1 im Graphen aus Abb. 8.1

Dieser Baum wird von den Algorithmen konstruiert. Die Wege lassen sich dann in Zeit proportional zur Anzahl der Kanten in den Wegen ausgeben.

Folgendes Lemma gibt einen Lösungsansatz, um aus einem k-kürzeste-Wege-Baum einen $(k + 1)$-kürzeste-Wege-Baum abzuleiten.

Lemma 8.2 *Seien T ein k-kürzeste-Wege-Baum für v_0 und w_{k+1} der $(k+1)$-kürzeste Weg $w_{k+1} = v_0 \cdots v_i v_{i+1} \cdots v_z$, dann ist $v_0 \cdots v_i$ ein Weg in T, der sich über die Kante $v_i v_{i+1}$ nicht in T verlängern lässt, und $v_{i+1} \cdots v_z$ ist der kürzeste Weg in G von v_{i+1} nach v_z.*

Beweis: Das i mit obiger Eigenschaft ist eindeutig durch T bestimmt, da nach Konstruktion an keinem Knoten v_i in T eine Kante $v_i v_{i+1}$ doppelt beginnen kann. Angenommen, $v_{i+1} \cdots v_z$ ist nicht kürzester Weg von v_{i+1} nach v_z, so gäbe es einen kürzeren Weg $w' = v_{i+1} \cdots v_z$ und somit einen Weg $w'_{k+1} = v_0 \cdots v_i v_{i+1} \overset{w'}{\rightsquigarrow} v_z$ mit $\gamma(w'_{k+1}) < \gamma(w_{k+1})$, der noch nicht in T ist, im Widerspruch zur Behauptung w_{k+1} sei der $(k+1)$-kürzeste Weg. □

Die Länge des Weges w_{k+1} ist allein durch die Kante $v_i v_{i+1}$ bestimmt. Der Weg selbst ist nur dann nicht eindeutig, wenn der kürzeste Weg von v_{i+1} zum Zielknoten nicht eindeutig ist.

Algorithmus 8.3 *Berechnung der k-kürzesten Wege von v_0 nach v_z*

Wir geben hier nur eine Algorithmusskizze: Zur Berechnung der k-kürzesten Wege benötigen wir den Kürzeste-Wege-Baum zum Zielknoten v_z. Dieser umfasst auch den kürzesten Weg w von v_0 nach v_z. Wir verwalten eine Menge K mit den Kandidaten für die nächstkürzesten Wege – zu Beginn ist $K := \{w\}$. In jedem der k Schleifendurchläufe wählen wir den kürzesten Weg aus K und fügen das nicht in T vorkommende Suffix zu T hinzu. Für jeden neu zu T hinzugefügten Knoten v implizieren die von dort ausgehenden Kanten vv' neue Kandidaten für den nächstkürzesten Weg. Aus den Kandidaten kann dann wieder der Weg mit kleinstem Gewicht ausgewählt und in T eingefügt werden.

Man beachte: Ein Knoten v kann mehrfach in T auftreten, sodass sich für die Kandidaten zu der Kante vv' gemerkt werden muss, um welches v in T es sich handelt. Jeder Kandidat lässt sich aber mit konstantem Platz durch die Kante und den Verweis zum Knoten in T eindeutig darstellen.

[MPS99] verwenden folgende Implementierungsdetails, um den Algorithmus effizient umzusetzen:

- Beobachtung 1: Man betrachte statt G den Graphen G_π, wobei wir als Potentialfunktion die kürzesten Entfernungen zum Zielknoten benutzen. Die k-kürzesten Wege sind nach Lemma 3.5 in G und G_π identisch und in G_π haben die kürzesten Wege zum Zielknoten nach Gleichung 3.4 die Länge 0. Dadurch ist die Weglänge der Kandidaten bereits durch

den in T liegenden Teil und der einen von T abzweigenden Kante gegeben.

- Beobachtung 2: Sortiert man in G_π von jedem Knoten v die ausgehenden Kanten in aufsteigender Reihenfolge des (reduzierten) Kantengewichts, so sind damit auch die Wege, die von v in T abzweigen, aufsteigend nach Länge sortiert.[2]

[MPS99] beschreiben einen $O(m \log n + k(n + \log k))$-Algorithmus, lassen dabei aber die Verwaltung der Menge K offen. Wir geben hier durch eine andere Implementierung eine verbesserte Laufzeitabschätzung trotz einfacher Datenstrukturen an.

Satz 8.4 *Die k kürzesten Wege in einem Graphen vom Startknoten v_0 zum Zielknoten v_z können in $O(m + n \log n + k(n + \log k))$ berechnet werden.*

Beweis: Die Berechnung des Kürzeste-Wege-Baums zum Zielknoten v_z, sowie die Berechnung von G_π können in $O(m + n \log n)$ durchgeführt werden.

Statt wie in [MPS99] die ausgehenden Kanten ($m_i \leq n$ für den Knoten v_i) in $\sum_{v_i \in V} m_i \log m_i \in O(m \log n)$ vorab zu sortieren, bilden wir lediglich für jeden Knoten v einen Heap $H_\pi(v)$ für die reduzierten Gewichte der ausgehenden Kanten, dies benötigt $\sum_{v_i \in V} m_i \in O(m)$. Jeder dieser Heaps $H_\pi(\cdot)$ wird im Laufe des Algorithmus zu einem teilsortierten Feld umgewandelt, sodass die ersten j Elemente sortiert vorliegen und die restlichen sich noch im Heap befinden.

In der i-ten Iteration wird aus den Kandidaten für den nächstkürzesten Weg ein Heap $H_{\text{next}(i)}$ aufgebaut. Aufgrund der Beobachtung 2 muss in diesem Heap für jeden (neuen) Knoten v in T nur der kürzeste von v ausgehende noch nicht betrachtete Weg gespeichert werden ($H_{\text{next}(i)}$ hat somit nur $O(n)$ Elemente). Erst wenn dieser als nächstkürzester Weg ausgewählt wird, muss der nächste von v in T abzweigende Weg wieder in den Heap $H_{\text{next}(i)}$ eingefügt werden. Merkt man sich für jedes v in T die zuletzt betrachtete Position in der Kantenliste von v, so ist der nächste Weg in $O(\log n)$ zu finden (entweder durch ein delete_min($H_\pi(v)$) in $O(\log n)$ oder, falls durch eine andere Kopie von v in T schon ein größerer Teil von $H_\pi(v)$ sortiert vorliegt, durch direkten Zugriff in $O(1)$).

Die Minima der Heaps $H_{\text{next}(\cdot)}$ werden in einem weiteren Heap $H_{\min}$ verwaltet, aus dem jeweils der nächstkürzeste Weg bestimmt wird. Die Anzahl

[2] Dies gilt separat für jedes Vorkommen von v in T.

der Elemente verringert sich in jeder Iteration um 1 bei der Auswahl des nächstkürzesten Weges (dieses wird ggf. durch das nächstgrößere Element des entsprechenden Heaps $H_{\text{next}(\cdot)}$ ersetzt) und erhöht sich um eins durch das Hinzufügen des Minimums des Heaps $H_{\text{next}(\cdot)}$ der aktuellen Iteration. Somit hat der Heap $H_{\min}$ nur $O(k)$ Elemente.

Der Aufwand pro Iteration lässt sich nun wie folgt abschätzen: Auswahl des nächstkürzesten Weges aus $H_{\min}$ in $O(\log k)$ – ggf. muss dadurch in einem der $H_\pi(v)$ der nächstgrößere Kandidat ermittelt und im entsprechenden $H_{\text{next}(\cdot)}$ eingefügt werden – Aufwand $O(\log n)$. Der ausgewählte nächstkürzeste Weg kann höchstens n neue Knoten zu T hinzufügen. Nach Lemma 8.2 bilden diese Knoten einen kürzesten und damit zyklenfreien Weg. Mit den jeweils kürzesten von diesen Knoten ausgehenden Kandidaten (diese stehen jeweils in der Wurzel der Heaps $H_\pi(\cdot)$ oder an erster Position im sortierten Teilfeld) lässt sich in $O(n)$ der neue Heap $H_{\text{next}(\cdot)}$ aufbauen und das Minimum in $O(\log k)$ in $H_{\min}$ einfügen. Der Aufwand in jeder der k Iterationen beträgt somit nur $O(n + \log k)$.

Die Gesamtlaufzeit ist damit bewiesen.[3] □

In fast allen uns vorliegenden k-kürzeste-Wege-Algorithmen wird k als klein gegenüber n angenommen und der Aufwand für die Verwaltung der Kandidaten und die Auswahl des nächstkürzesten Weges vernachlässigt. Diese benötigt bei dem iterativen Vorgehen mindestens $O(k \log k)$, da die k kleinsten Kandidaten sortiert ausgegeben werden.

Dies trifft nicht auf den verbesserten Algorithmus nach [Epp98] zu. Dieser berechnet unter Ausnutzung weiterer Eigenschaften und komplizierterer Datenstrukturen in einer Laufzeit von $O(m + n \log n + k)$ für Graphen mit positiven Kantengewichten eine implizite Darstellung der k kürzesten Wege, sodass diese in Zeit proportional zur Anzahl der Kanten ausgegeben werden können.[4] Die Idee ist eine Erweiterung der Beobachtung 2 mit der [Epp98] einen Heap aufbaut, der die k-kürzesten Wege implizit umfasst. Mit der Methode von [Fre93] lassen sich die k kleinsten Elemente dieses Heaps in $O(k)$ entnehmen, diese liegen dann jedoch nicht sortiert vor.

[3] Im vergleichsbasierten Modell ist diese Laufzeit optimal, wenn der k-kürzeste-Wege-Baum so explizit angegeben werden soll (dieser hat $O(nk)$ Knoten). Der kürzeste Weg wird berechnet (Aufwand $O(m + n \log n)$, optimal nach Satz 3.38) und die k kürzesten Wege werden in sortierter Reihenfolge ausgegeben (Aufwand $\Omega(k \log k)$ im Worst-Case).

[4] Die Datenstruktur ist eine kompaktifizierte Darstellung des k-kürzeste-Wege-Baums. Der j-kürzeste Weg wird als Verweis auf den Index i des i-kürzesten Weges und der von dort abzweigenden Kante gespeichert, sodass der Speicherplatz gesamt nur $O(n + k)$ beträgt. Die obige Darstellung kann bis zu $O(kn)$ Knoten enthalten, sodass die Laufzeit von Eppsteins Algorithmus [Epp98] nicht erreicht werden könnte.

8.2 Das eingeschränkte k-kürzeste-Wege-Problem

Die Algorithmen von Yen und Lawler ([Yen71], [Law72]) lösen das Problem, die k kürzesten zyklenfreien Wege zu berechnen. Gegenüber obigem Algorithmus muss bei der Bestimmung der Kandidaten beachtet werden, dass nach dem Abzweigen aus T keine Knoten des Weges aus T wiederholt werden. Dazu werden in jeder der k Iterationen bis zu n SSSP-Probleme in entsprechend ausgedünnten Graphen gelöst. Die Laufzeit resultiert dann in $O(kn(m + n \log n))$. Für ungerichtete Graphen geben [KIM82] die verbesserte Schranke $O(k(m + n \log n))$ an. Auch hier gilt obige Anmerkung, dass die Laufzeit $O(k \log k)$ größer ist (die in den Artikeln angegebenen Methoden zur Verwaltung der Menge der Kandidaten haben noch etwas größeren Aufwand). Über eine Heuristik erreichen [HMS03] in gerichteten Graphen einen Geschwindigkeitsgewinn von einem Faktor $\Theta(n)$. Experimentell ermittelt funktioniert die Heuristik in über 99% der Fälle. Schlägt sie fehl, kann dies leicht festgestellt werden und wieder auf die Algorithmen von Yen und Lawler zurückgegriffen werden.

8.3 Weitere Aussagen zu k-kürzeste-Wege-Problemen

Die oben angegebenen Laufzeiten beziehen sich alle auf Graphen mit positiven Kantengewichten. Über Potentialfunktionen lassen sich die Algorithmen leicht auf Graphen mit negativen Kantengewichten erweitern.

Satz 8.5 *Sei A ein k-kürzeste-Wege-Algorithmus für Graphen mit positiven Kantengewichten und Laufzeit $O(f(n, m, k))$, so lässt sich mit A in beliebigen Graphen (ohne negative Zyklen) das k-kürzeste-Wege-Problem in $O(nm + f(n, m, k))$ Schritten lösen.*

Beweis: Man berechne wie in Lemma 3.14 eine Potentialfunktion $\pi(\cdot)$, sodass im reduzierten Graphen G_π die Kantengewichte positiv sind. Nach Lemma 3.5 bleibt die Ordnung der Weglängen erhalten, sodass die k kürzesten Wege in G und G_π identisch sind. □

In azyklischen Graphen sind beide Problemvarianten (mit und ohne erlaubte Zyklen) identisch, sodass hier stets die schnelleren Algorithmen für das uneingeschränkte k-kürzeste-Wege-Problem zum Einsatz kommen können. Die k-kürzesten Wege sind aufgrund des Ausgangsgraphen stets zyklenfrei.

Ein Problem bei der k-kürzesten-Wege-Suche in der Anwendung, z.B. bei Navigationssystemen, ist, dass die nächstkürzesten Wege sich oft nur minimal vom kürzesten Weg unterscheiden (z.B. an einer Autobahnausfahrt abfahren und an derselben gleich wieder auf die Autobahn auffahren), während in der Anwendung eher wirkliche Alternativrouten gesucht sind, um z.B. Staus zu umfahren. [SII+95] untersuchen dieses Problem, bei dem die Alternativen nur eingeschränkt mit dem kürzesten Weg übereinstimmen dürfen.

Kapitel 9

Übersicht über weitere Ansätze und Problemstellungen

9.1 Algorithmen für planare Graphen

Planare Graphen bilden eine wichtige Graphenklasse bzgl. Kürzeste-Wege-Probleme. Die asymptotisch effizientesten Algorithmen für planare Graphen lassen sich sehr kurz zusammenfassen. [Fre87] löst das APSP-Problem in planaren Graphen mit nichtnegativen Kantengewichten optimal in $O(n^2)$. Da mit Hilfe des Bellman-Ford-Algorithmus ein SSSP-Problem in planaren Graphen wegen $m \in O(n)$ in $O(n^2)$ gelöst werden kann, kann mit Hilfe der damit berechneten Potentialfunktion aus Lemma 3.14 das APSP-Problem in planaren Graphen mit beliebigen Kantengewichten ohne asymptotischen Mehraufwand gelöst werden. [HKRS97] geben für planare Graphen mit nichtnegativen Kantengewichten einen Linearzeitalgorithmus an und [Kle04] beschreibt für planare Graphen einen Algorithmus mit Laufzeit $O(n \log n)$. Alle diese Algorithmen arbeiten im Wesentlichen auf einem vergleichsbasierten Modell.

Der Dijkstra-Algorithmus (3.30 mit Lemma 3.34) löst das SSSP-Problem in Graphen mit nichtnegativen Kantengewichten mit Heaps in $O(n \log n)$. Da die Konstante im $O(\cdot)$ sehr klein ist, ist für die Praxis dieser Algorithmus sehr gut geeignet. Die beiden im Folgenden vorgestellten Algorithmen sind hauptsächlich von theoretischem Interesse, für die Praxis sind die Konstanten im $O(\cdot)$ zu groß.

Zerfällt ein Graph durch Entfernen weniger Knoten in mehrere Zusammenhangskomponenten, so kann man mit Divide-and-Conquer-Ansätzen versuchen, die Probleme in den Untergraphen zu lösen und die Teillösungen zu

einer Gesamtlösung zusammenzufügen. In planaren Graphen gibt es solche Separatoren mit relativ wenig Knoten.

Definition 9.1 (Separator) *Eine unter Untergraphenbildung abgeschlossene Graphenklasse $\mathcal{G}$, d.h., falls $G_1 \in \mathcal{G}$ und G_2 Untergraph von G_1 ist, so gilt auch $G_2 \in \mathcal{G}$, erfüllt ein* $f(n)$-Separatortheorem *genau dann, wenn für $\alpha < 1$ für jeden Graphen $G = (V, E) \in \mathcal{G}$ eine Zerlegung $V = A \cup B \cup S$ mit $|S| \in O(f(n))$, $|A|, |B| \leq \alpha n$ und $(A \times B \cup B \times A) \cap E = \emptyset$ existiert.*

In der Regel betrachtet man Separatortheoreme mit $\alpha = \frac{2}{3}$. Man beachte jedoch, dass ein Separatortheorem für $\alpha = 1 - \varepsilon$ durch rekursive Anwendung auf die entstehenden Komponenten eines für $\alpha = \frac{1}{2}$ impliziert! Die Größe des Separators steigt dabei nur um einen konstanten Faktor an.

Für planare Graphen wurde zunächst von [LT79] und [LT80] das folgende Separatortheorem gezeigt.

Satz 9.2 (Separatortheorem für planare Graphen) *Jeder planare Graph $G = (V, E)$ hat eine Zerlegung $V = A \cup B \cup S$ mit $|S| \leq 2\sqrt{2}\sqrt{n}$, $|A|, |B| \leq \frac{2}{3}n$ und $(A \times B \cup B \times A) \cap E = \emptyset$. Der Separator S lässt sich in Linearzeit konstruieren.*

Die Konstante $2\sqrt{2}$ wurde von [DV97] auf 2, in ungerichteten Graphen sogar auf $\sqrt{2/3} + \sqrt{4/3} \approx 1.97$ reduziert (die Konstruktion erfolgt ebenfalls in Linearzeit), ebenfalls von Djidjev ([Dji82]) stammt die untere Schranke $\frac{1}{3}\sqrt{4\pi\sqrt{3}}\sqrt{n} \approx 1.56\sqrt{n}$.

Die Algorithmen von [Fre87] und [HKRS97] basieren auf Separatoren, sie sind jedoch sehr komplex. Wir geben daher hier nur sehr kurze Beschreibungen an. Der Algorithmus von [Fre87] löst in planaren Graphen das SSSP-Problem in $O(n\sqrt{\log n})$ (eine detaillierte Darstellung findet sich z.B. in [Lew97]). Dazu wendet dieser die Separatortheoreme von [LT79] und [LT80] rekursiv an und erhält so eine Aufteilung in Regionen mit je logarithmisch vielen Knoten. Diese Aufteilung lässt sich so berechnen, dass der Großteil der Knoten untereinander und nur wenige mit Knoten aus den Separatoren verbunden sind. Jede Region wird dann nochmals in noch kleinere Parzellen aufgeteilt. Zwischen denen zu einer Parzelle angrenzenden Separatorknoten werden dann jeweils paarweise die kürzesten Verbindungen berechnet. Durch die geringe Anzahl der Knoten in diesen Parzellen bleibt der Aufwand gering. Mit Hilfe einer auf diese zweistufige Zerlegung ausgerichteten Datenstruktur kann [Fre87] dann das SSSP-Problem in $O(n\sqrt{\log n})$ lösen.

Mit einer dreistufigen Zerlegung und einem ähnlichen Ansatz löst [Fre87] das APSP-Problem optimal in $O(n^2)$.

Ebenfalls auf solchen mehrstufigen Separatorzerlegungen baut der Algorithmus von [HKRS97] auf. Während [Fre87] entfernungssetzend arbeitet (vgl. Abschnitt 3.4), ist der Algorithmus von [HKRS97] entfernungskorrigierend (vgl. Abschnitt 3.3), d.h. die von den Knoten ausgehenden Kanten müssen ggf. mehrfach betrachtet werden. Aufgrund der Eigenschaften der Separatorzerlegung und einer geschickten Aufteilung der decrease_key(·)-Aufrufe bleibt die Laufzeit aber sogar linear.

Während in allgemeinen Graphen das SSSP-Problem für Graphen mit beliebigen Kantengewichten gegenüber dem für nichtnegative Kantengewichte deutlich schwieriger ist, gibt [Kle04] einen effizienten Algorithmus an, der das Problem auf planaren Graphen mit beliebigen Kantengewichten in $O(n \log n)$ löst (oder einen negativen Zyklus erkennt) und somit nur wenig langsamer als der Algorithmus von [HKRS97] für planare Graphen mit nichtnegativen Kantengewichten ist. Während [Fre87] und [HKRS97] nur die Separatoreigenschaften ausnutzen, basiert der Algorithmus von [Kle04] sehr stark auf die Einbettung des planaren Graphen in die Ebene.

9.2 Algorithmen für fast azyklische Graphen

Für azyklische Graphen haben wir in Abschnitt 3.4.1 einen Linearzeit-Algorithmus vorgestellt. Dies lässt hoffen, dass für Graphen, die nach Entfernen weniger Kanten azyklisch wären, schnellere Algorithmen als für allgemeine Graphen existieren.

[AK94] stellen einen Algorithmus vor, der ähnlich wie der Linearzeitalgorithmus aus Abschnitt 3.4.1 funktioniert. Solange Knoten ohne noch nicht betrachtete eingehende Kanten existieren, werden diese wie im azyklischen Fall behandelt. Nur sonst werden Knoten über delete_min(·) ausgewählt. Die D(·)-Werte werden mit einer Variante der Fibonacci-Heaps verwaltet. Sind im Laufe des Algorithmus $\hat{n} \leq n$ delete_min(·)-Operationen nötig, so ist die Laufzeit $O(m + n \log \hat{n})$. Die im $O(\cdot)$ versteckte Konstante ist vergleichbar mit der bei normalen Fibonacci-Heaps, sodass diese Variante nie wesentlich langsamer als die Dijkstra-Implementierung mit Fibonacci-Heaps ist. Bei azyklischen Graphen ist $\hat{n} = 0$ und die Laufzeit linear. [AK94] erwarten für fast azyklische Graphen, dass $\hat{n}$ klein gegenüber n ist, ohne dies jedoch mit weiteren Argumenten belegen zu können.

[Tak98a] nehmen die Anzahl der Knoten der größten starken Zusammenhangskomponente n_{cyc} als Maß für die Azyklizität. Bei azyklischen Graphen ist dieses Maß 1, bei stark zusammenhängenden Graphen n. Der Algorithmus basiert auf zwei Beobachtungen:

1. Fasst man die starken Zusammenhangskomponenten jeweils zu einem Knoten zusammen, so ist der entstehende Graph azyklisch.

2. Initialisiert man die SSSP-Suche so, dass die Knoten u einer Teilmenge $U \subseteq V$ mit $\mathrm{D}(u) = 0$ belegt werden, so geben die berechneten Werte die kürzeste Entfernung zu dem jeweils nächstgelegenen Knoten aus U an, analog kann man die Knoten $u \in U$ mit einem Potential belegen, sodass die kürzesten Wege unter Berücksichtigung dieser Potentiale berechnet werden.

Das Vorgehen ist nun wie folgt: Die starken Zusammenhangskomponenten werden bzgl. des zugrunde liegenden azyklischen Graphen (obige Beobachtung 1) topologisch sortiert und nun in dieser Reihenfolge bearbeitet. Die aus der starken Zusammenhangskomponente hinausführenden Kanten setzen die Potentiale für die SSSP-Suche innerhalb der nächsten zu untersuchenden starken Zusammenhangskomponenten. Neben dem linearen Aufwand für die Berechnung der topologischen Sortierung und dem azyklischen Teil des Algorithmus bleibt die Summe für die SSSP-Suchen innerhalb der starken Zusammenhangskomponenten. Haben diese $n_1, \ldots, n_k$ Knoten, so folgt der Gesamtaufwand $O(m + n + \sum_{i=1}^{k} (m_i + n_i \log n_i))$. Da jede Kante nur in einer starken Zusammenhangskomponente beginnen kann, lässt sich dies mit $n_i \leq n_{\mathrm{cyc}}$ zu $O(m+n+\sum_{i=1}^{k} n_i \log n_{\mathrm{cyc}}) = O(m+n \log n_{\mathrm{cyc}})$ zusammenfassen.

Verwendet man den Algorithmus von [AK94] statt der normalen SSSP-Suche innerhalb der starken Zusammenhangskomponenten, ergibt sich ein Aufwand von $O(m + n \log \tilde{n})$ mit $\tilde{n} \leq \min\{\hat{n}, n_{\mathrm{cyc}}\}$.

[ST03] schlagen vor, den Graphen in azyklische Teilgraphen zu zerlegen. Ist so eine Zerlegung in r Teilgraphen gegeben, so lässt sich das SSSP-Problem in $O(m + r \log r)$ lösen. Die Berechnung einer optimalen Zerlegung (mit minimalem r) benötigt jedoch zusätzlich $O(nm)$.

Die obigen Algorithmen arbeiten nur auf Graphen mit positiven Kantengewichten. Wir beschreiben nun noch kurz eine modifizierte Variante des Bellman-Ford-Algorithmus nach Goldberg und Radzik ([GR93]), die in beliebigen Graphen den Worst-Case von $O(nm)$ beibehält, aber in azyklischen Graphen die kürzesten Wege in Linearzeit berechnet. Die Grundidee des Algorithmus ist die Beobachtung, dass – wenn die Kante uv verkürzend ist – man den Knoten u vor dem Knoten v betrachten sollte, weil sich $\mathrm{D}(v)$ beim Betrachten des Knotens u verringern wird. Um dies zu erreichen arbeiten [GR93] in Phasen ähnlich wie im Bellman-Ford-Algorithmus. In jeder Phase wird die Menge M der Knoten betrachtet, deren $\mathrm{D}(\cdot)$-Werte in der vorhergehenden Phase reduziert wurden. Ähnlich wie in Goldbergs Skalierungsalgorithmus aus Abschnitt 6.2 wird nun ein Graph $G_{\pi \leq 0, sZ}$ betrachtet: Zunächst

werden die Knoten aus M entfernt, die keine ausgehenden verkürzenden Kanten haben. Nun betrachten [GR93] die von den verbleibenden Knoten aus M über Kanten mit $\gamma_\pi(\cdot) \leq 0$, $\pi(\cdot) := \mathrm{D}(\cdot)$ erreichbaren Knoten. In diesem Teilgraphen werden wie in Abschnitt 6.2 die starken Zusammenhangskomponenten berechnet, ggf. Komponenten auf einen Knoten zusammengezogen und in dem azyklischen Graphen $G_{\pi \leq 0, sZ}$ in der Reihenfolge einer topologischen Sortierung der Knoten aus M bzgl. $G_{\pi \leq 0, sZ}$ die Potentialfunktion und damit die $\mathrm{D}(\cdot)$-Werte in G angepasst. Die Korrektheit folgt ähnlich wie bei Bellman-Ford: Nach i Phasen sind mindestens die i ersten Ebenen des Kürzeste-Wege-Baums korrekt berechnet.

In der Praxis erweist sich diese Variante als sehr robust und ist bei der umfangreichen Untersuchung von [CGR96] zwar nicht für viele Graphenklassen der beste Algorithmus, er verliert aber gegenüber den besten Algorithmen nur einen kleinen Faktor. Sind die Eigenschaften der Graphen a priori nicht bekannt, ist dieser Algorithmus also eine sehr gute Wahl. Kein anderer Algorithmus war in den Untersuchungen von [CGR96] ähnlich robust!

9.3 Geometrische Beschleunigungsverfahren

In diesem Abschnitt wird ein Ansatz von [SWW00], [Sch00b] und [WW03] vorgestellt, mit dem die Anzahl der bei der kürzesten Wege Suche betrachteten Kanten deutlich reduziert werden kann. Der Ansatz ist kein eigenständiger Algorithmus sondern vielmehr eine Beschleunigung, die bei fast allen SSSP-Algorithmen angewendet werden kann, wenn nur der kürzeste Weg zu *einem* Zielknoten gesucht wird (SPSP-Problem).

Die Idee ist dabei, nicht alle ausgehenden Kanten eines Knotens zu betrachten, sondern nur die, die auf dem kürzesten Weg liegen könnten. Dazu werden in einer Preprocessingphase für jeden Knoten u zu jeder Kante uv die Menge der Knoten gespeichert, zu denen ein kürzester Weg führt, der mit dieser Kante uv beginnt. Im Ablauf des Kürzeste-Wege-Algorithmus werden dann die Kanten ignoriert, für die der Zielknoten nicht in der vorberechneten Menge liegt. Das Abspeichern aller Knoten an jeder Kante würde gesamt $O(nm)$ Platz benötigen – mehr als die $O(n^2)$ für das Abspeichern aller kürzesten Entfernungen. Deshalb beschränken sich die Autoren darauf, mit $O(1)$ Platz pro Kante eine Obermenge der Knoten zu beschreiben, die über diese Kante auf einem kürzesten Weg zu erreichen sind.

Sie untersuchen dazu unter anderem Winkelsektoren, Kreise, Rechtecke in verschiedenen Varianten. In Experimenten hat sich dabei das einfache um-

fassende Rechteck als im Mittel am Besten geeignet herausgestellt. Es beschreibt die Knotenmengen zwar nicht immer optimal, ist aber sehr effizient zu überprüfen. Der Dijkstra-Algorithmus wurde dadurch in den Experimenten auf realen Straßen- bzw. Eisenbahngraphen um Faktoren von etwa 10 bis 20 beschleunigt.

Eine Kombination mit zielgerichteter Suche (z.B. A^*-Heuristik, Abschnitt 3.4.4) oder beidseitiger Suche (Kapitel 4) ist möglich.

Obwohl in der Praxis im Mittel eine deutliche Beschleunigung erreicht wird, wird der Worst-Case durch die Verfahren nicht verbessert. Da die Beschreibung der Knotenmenge auf geometrischen Eigenschaften des Graphen basiert, lassen sich stets leicht Beispiele konstruieren, sodass in jedem Schritt alle ausgehenden Kanten betrachtet werden müssen – z.B. Graphen, bei denen die Wege wie in einem Schneckenhaus spiralförmig verlaufen, sodass eine einfache geometrische Beschreibung z.B. mit Rechtecken nicht möglich ist.

9.4 Der Component-Tree

Die in Abschnitt 5.4 beschriebenen Ideen von [Din78] hat [Tho99] rekursiv angewendet und somit ein Linearzeitverfahren für das SSSP-Problem mit ganzzahligen positiven Kantengewichten in ungerichteten Graphen entwickelt.

Ähnlich wie im Kaliberlemma 5.8 sagt Thorup aus, dass, falls zwei Knotenmengen nur durch Kanten mit Gewicht $\gamma(\cdot) \geq c$ verbunden sind, in der einen Knotenmenge für Knoten ggf. schon $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ garantiert werden kann, auch wenn diese Werte um bis zu c größer als $\min(\mathrm{D}(R))$ sind.

Im Prinzip wird dabei der Graph bzgl. der Binärdarstellung der Kantengewichte separiert. Der Component-Tree ist im Wesentlichen ein minimaler Spannbaum. Sind im Spannbaum zwei Knoten nur durch Kanten mit $\gamma(\cdot) > c$ verbunden, so gilt dies auch im Graphen selbst. Thorup teilt den Graphen damit in Knotenmengen ein, die untereinander jeweils mit Kanten mit $\gamma(\cdot) \leq 2^i$ verbunden sind. Die Kürzeste-Wege-Suche wird so zur Traversierung dieses Component-Tree.

Die Verwendung von Atomic Heaps ([FW94]), die erst ab $n > 2^{12^{20}}$ definiert sind, lässt dieses aber als ein rein theoretisches Resultat erscheinen. [AI00] und [Hag00] schlagen Änderungen vor, um die Beschränkung auf solche unrealistischen Größen unter Beibehaltung der Linearzeit im Worst-Case zu beseitigen. So implementiert sind die Konstanten im $O(\cdot)$ der Laufzeit aber so groß, dass [AI00] für die Praxis weiterhin keine (sinnvolle) Anwendung sieht.

Interessant ist der Component-Tree insofern, dass man ihn als Preprocessing auffassen kann – er ist unabhängig vom Start/Zielknoten. [Pet02], [Pet04] geben z.B. einen SSSP-Algorithmus im vergleichsbasierten Modell für reellwertige positive Kantengewichte an, der nur $O(m+n \log\log n)$ Schritte benötigt, wenn der Component-Tree bereits gegeben ist. Daher ist der Component-Tree-Ansatz auch interessant, wenn man mehrere kürzeste Wege berechnen will, Experimente in [PRS02] bestätigen dies.

Obwohl dieser Ansatz im RAM-Modell auf ungerichteten Graphen zu einem optimalen $O(m)$-Algorithmus geführt hat, hilft er bei der Suche nach einem besseren vergleichsbasierten SSSP-Algorithmus nicht weiter. [Pet04] (auch [Pet03]) zeigt selbst für den ungerichteten Fall eine untere Schranke von $\Omega(m + n \log n)$ Schritten für die Berechnung des Component-Tree, sodass dieser Ansatz im vergleichsbasierten Modell asymptotisch nicht schneller als der Dijkstra-Algorithmus mit Fibonacci-Heaps sein kann.

9.5 Ausblick

Die verfügbare Literatur zu Kürzeste-Wege-Algorithmen scheint uferlos. Ohne zu sehr ins Detail zu gehen seien hier noch weitere Problemstellungen und Ansätze samt Literaturverweisen gegeben.

Average-Case-Verhalten von Kürzeste-Wege-Algorithmen

Theoretisch fundierte Untersuchungen zum Average-Case-Verhalten von Kürzeste-Wege-Algorithmen gibt es vergleichsweise wenig. Neben den in Abschnitt 5.4 genannten Average-Case-Linearzeit-Algorithmen von Goldberg, Hagerup und Meyer ([Gol04], [Hag04], [Mey01]) und den im Abschnitt 7.3 genannten APSP-Verfahren ([HZ85], in Verbindung mit [KKP93] und [McG95], sowie [MT87]), seien dem interessierten Leser hier noch die Arbeiten von Priebe ([Pri01], [CFMP00]) und Meyer ([Mey02], [Mey03]) genannt, die obere und untere Schranken für das Average-Case-Verhalten einiger Kürzeste-Wege-Algorithmen beweisen und auf weitere Artikel verweisen.

Algorithmen mit Preprocessing und sublinearen Antwortzeiten

Speziell in Anwendungen wie Navigationssystemen oder Fahrplanauskunftssystemen wird man auf demselben Graphen viele Kürzeste-Wege-Berechnungen durchführen wollen. Eine Möglichkeit wäre, mit einem APSP-Algo-

rithmus alle Wege zu berechnen und sich in $O(n^2)$ Platz die Kürzeste-Wege-Bäume und Entfernungen zu merken. Die kürzesten Entfernungen können dann optimal in $O(1)$ und der kürzeste Weg w in $O(|w|)$ ausgegeben werden. Dieser Platz steht in der Regel nicht zur Verfügung, sodass Datenstrukturen entwickelt werden müssen, die mit möglichst wenig Platz die Kürzeste-Wege-Suche möglichst effizient beschleunigen.

Es ist hier im Wesentlichen zu unterscheiden, ob exakte Lösungen erforderlich sind oder Näherungslösungen ausreichen. Oft werden dabei hierarchische Ansätze verfolgt. Viele Veröffentlichungen bleiben dabei auf einem sehr ingenieurmäßigen Level – oft wird nicht einmal angegeben, wie gut oder schlecht die gefundenen Lösungen sind. [Buc00] beschreibt einige theoretisch fundierte Ansätze und gibt weitere Literaturhinweise. Einige neuere Ansätze sind in [DPZ00], [Kle02] und [Tho01] für planare Graphen und in [EP04], [Hol03], [PSWZ04], [SWZ02] und [TZ01] für allgemeine Graphen zu finden. Weitere Literaturhinweise gibt auch [Zwi01].

Dynamische Verfahren zur Kürzeste-Wege-Berechnung

In vielen Anwendungen, z.B. bei Navigationssystemen, ergeben sich Änderungen der Kantengewichte (z.B. durch temporäre Geschwindigkeitsbegrenzungen oder Sperrung von Straßenabschnitten). Der Natur nach handelt es sich auch hier um Verfahren, die in einer Preprocessingphase Datenstrukturen aufbauen, um danach kürzeste Wege schnell berechnen zu können. Waren im vorhergehenden Abschnitt die Datenstrukturen statisch, so sollen hier nun zusätzlich Änderungen in den Graphen in diesen Datenstrukturen schnell nachvollzogen werden, sodass die Preprocessingphase nicht von null neu beginnen muss.

Für planare Graphen gibt [DPZ00] Algorithmen an. [AMO93] gibt in den Übungen zu den Kapiteln 4 und 5 Literaturhinweise wie sich die kürzesten Wege verändern, wenn sich Kantengewichte ändern oder Kanten hinzugefügt bzw. gelöscht werden.

Weitere

Die Liste ließe sich fast endlos weiterführen, es seien hier nur noch einige Beispiele genannt.

Die meisten hier vorgestellten Algorithmen sind sehr auf sequentielle Verarbeitung ausgerichtet. Eine parallele Verarbeitung ist nur an wenigen Stellen möglich (z.B. beim Betrachten der von einem Knoten ausgehenden Kanten).

Die Arbeiten von Meyer ([Mey02] und [MS03]) geben neben weiteren Literaturhinweisen einen guten Einblick in das Gebiet der parallelen Kürzeste-Wege-Berechnung.

In Bäumen sind Wege eindeutig – damit auch die kürzesten Wege. Nach einer Preprocessingphase mit linearer Laufzeit und linearem Speicherbedarf können die kürzesten Entfernungen in konstanter Zeit und die zugehörigen Wege w in $O(|w|)$ ausgegeben werden ([Buc00]). Dies motiviert Algorithmen, die in „baumähnlichen"Graphen effizient kürzeste Wege berechnen. Die Arbeiten von Bodlaender ([Bod93], [Bod97]) geben einen guten Überblick über Eigenschaften und Resultate bzgl. Graphen mit kleiner „Baumweite". Die Arbeiten von Chaudhuri et.al. ([CZ00], [CZ98]) beschäftigen sich speziell mit sequentiellen und parallelen Algorithmen zur Kürzeste-Wege-Berechnung in solchen Graphen.

[BCKM01] geben eine Prioritätswarteschlange für ganzzahlige positive Gewichte an, die $O(1)$ Aufwand für alle (hier benötigten) Operationen hat – somit ist die Kürzeste-Wege-Berechnung in Linearzeit möglich. Die Datenstruktur benötigt aber ein nicht standardmäßiges Speichermodell: Normalerweise besteht eine Speicherzelle aus einer Folge von Bits, die nur in dieser Speicherzelle vorkommen. Bei dem von [BCKM01] verwendeten Modell können einzelne Bits Teil mehrerer Speicherzellen sein. Wird das Bit an einer Stelle geändert, ändert es sich an anderen automatisch mit, sodass es auf heutigen Rechnern keine effiziente Implementierung der Datenstruktur gibt. Eine Festverdrahtung als Hardware ist jedoch denkbar.

Bei Fahrplanauskünften ergeben sich zusätzliche Anforderungen an die Modellierung: Zum einen hängt die Suche von dem Zeitpunkt ab, für den eine Verbindung gesucht wird, zum anderen möchte man dabei die Verbindung bestimmen, die bei gleicher Ankunftszeit möglichst spät abfährt. [PS97] geben einen Überblick über Algorithmen in diesem Bereich. [PSWZ04] und [SWZ02] versuchen mit hierarchischen Ansätzen effiziente Algorithmen hierfür zu entwickeln.

In manchen Anwendungen ist der normale Begriff des Graphen nicht mehr geeignet: Sollen z.B. Roboter sich in der Ebene bewegen, so bestimmen Hindernisse die möglichen Bewegungen. [HS99] und [CM99] beschreiben Algorithmen, um in solch einem Szenario kürzeste Wege zwischen zwei Punkten in der Ebene unter Berücksichtigung von Hindernissen – gegeben als geometrische Objekte – zu berechnen. [Mit00] gibt in seiner umfangreichen Arbeit einen Überblick für diverse Varianten der Kürzeste-Wege-Berechnung in der Ebene an – das Literaturverzeichnis umfasst 395 Einträge.

9.6 Offene Probleme

Im Laufe dieser Arbeit sind einige Ideen und Fragestellungen entstanden, die noch nicht zu veröffentlichungsreifen Ergebnissen geführt werden konnten.

Beim generischen Algorithmus 3.18 wurde eine Schranke von $O(2^{n \log n})$ Iterationen bewiesen, dieses lässt sich unter Berücksichtigung der Kantenanzahl auf $O(2^{n \log (m/n)})$ verbessern. [AMO93] behaupten eine Schranke von $O(2^n)$ Iterationen, beziehen sich dabei aber auf [GP86], die diese Schranke nur für die modifizierte Variante angeben. Fügt man in die Worst-Case-Beispiele zum D'Esopo-Pape-Algorithmus (Abbildungen 3.5 bis 3.8) auch Kanten von größeren zu kleineren Indizes hinzu, so kann man Beispiele konstruieren, die doppelt soviele Iterationen benötigen wie der Worst-Case des modifizierten generischen Algorithmus. Die obere Schranke ist dann aber immer noch $O(2^n)$. Es ist uns nicht bekannt, ob ein Beweis für $O(2^n)$ Iterationen für den generischen Algorithmus geführt werden kann oder ob es Beispiele mit $\omega(2^n)$ Iterationen gibt.

In Lemma 3.14 haben wir eine konsistente Potentialfunktion in $O(nm)$ konstruiert. Lässt sich auf einem anderen Weg eine konsistente Potentialfunktion in $o(nm)$ konstruieren, die die kürzesten Entfernungen ggf. auch unterschätzen darf (vgl. Bemerkung 3.12), so wäre eine Berechnung der kürzesten Wege in Graphen G mit beliebigen Kantengewichten in $o(nm)$ möglich, indem statt in G die kürzesten Wege in G_π – $\gamma_\pi(\cdot) \geq 0$, da $\pi(\cdot)$ konsistent – berechnet werden. Die Wahlmöglichkeiten für die Potentialfunktionen π sind innerhalb der starken Zusammenhangskomponenten nicht groß, da nach Lemma 3.3 die Gewichte von Zyklen gleich bleiben – hat ein Zyklus in G Länge 0, so müssen die reduzierten Kantengewichte in G_π auf diesem Zyklus alle 0 sein, d.h., wird $\pi(\cdot)$ für einen Knoten des Zyklus festgelegt, sind damit auch die Werte $\pi(\cdot)$ für die anderen Knoten des Zyklus bestimmt.

Im selben Zusammenhang stellt sich die Frage, ob die Potentialfunktion in Lemma 3.14 bestimmte Eigenschaften erfüllt, z.B. die Maximierung der reduzierten Kantengewichte oder ähnliches. Innerhalb starker Zusammenhangskomponenten sind solche Eigenschaften denkbar. Sonst kann für die Kante uv z.B. das Potential $\pi(v)$ beliebig klein gewählt werden, sodass $\gamma_\pi(uv)$ beliebig groß wird.

Beim Algorithmus von D'Esopo und Pape (Abschnitt 3.3.2) haben wir angemerkt, dass der Algorithmus in Graphen mit häufig verletzter eingeschränkter Dreiecksungleichung, d.h. $\gamma(uw) > \gamma(uv) + \text{dist}(v, w)$, zu langen Laufzeiten tendiert, da nach dem Bearbeiten der von u ausgehenden Kanten die Schritte von w aus umsonst ausgeführt werden, da später $\text{D}(w)$ über den Weg über

v wieder verringert wird. Ist $\gamma(uw) \leq \gamma(uv) + \mathrm{dist}(v, w)$ für alle $u, v, w \in V$ erfüllt, so kann jede Kante im Graphen auch Teil eines kürzesten Weges sein – dies ist bei den Worst-Case-Beispielen nicht der Fall gewesen. Wir vermuten, dass die Laufzeit dann polynomiell beschränkt ist. Die von uns konstruierten Beispiele haben nie mehr als quadratisch viele Iterationen benötigt.

Beim Dijkstra- und A^*-Algorithmus werden die Knoten bzgl. $\mathrm{d}(\cdot)$ bzw. $\mathrm{d}(\cdot) + \widehat{\mathrm{ed}}(\cdot)$ in monoton aufsteigender Reihenfolge ausgewählt. Bei Auswahl nach Dinitz ist dieses bzgl. $\lfloor \mathrm{d}(\cdot)/\gamma_{\min} \rfloor$ der Fall. Dies lässt folgende Vermutung zu: Erfolgt die Auswahl eines Knotens aus R aufgrund der Wahl des Minimums von $f(\mathrm{D}(R))$, so gilt für die ausgewählten Knoten $\mathrm{D}(\cdot) = \mathrm{d}(\cdot)$ genau dann, wenn die Werte $f(\mathrm{D}(\cdot))$ der ausgewählten Knoten eine monoton steigende Folge bilden. Gilt dies für beliebige Funktionen $f(\cdot)$?

Der Skalierungsalgorithmus nach [Gol95] nutzt in dem SSSP-Algorithmus für $\gamma(\cdot) \geq -1$ nicht aus, dass die kürzesten Wege alle Länge von höchstens 0 haben. Lässt sich dies z.B. über Ansätze wie bei den azyklischen Graphen ausnutzen? Die $\sqrt{n}$ Iterationen, die [Gol95] braucht, lassen gegenüber der einen benötigten Iteration, falls die kürzesten Wege alle nur Kanten mit $\gamma_\pi(\cdot) \in \{-1, 0\}$ benutzen, scheinbar viel Spielraum für Verbesserungen – leider konnten zu jedem Ansatz bisher Gegenbeispiele konstruiert werden.

Kapitel 10

Zusammenfassung und Ausblick

Wir haben über den generischen Kürzeste-Wege-Algorithmus die meisten geläufigen Algorithmen hergeleitet und durch die gezeigten Zusammenhänge vereinfacht bewiesen.

Wir haben gezeigt, dass der klassische Ansatz bei der beidseitigen heuristischen Suche zur exakten Lösung des SPSP-Problems keine Vorteile bringen kann. Im Gegensatz dazu bringt die beidseitige heuristische Suche aber schnell beweisbar gute Näherungen – diese dürften in der Praxis nicht selten sogar den tatsächlich kürzesten Weg finden.

Die Bucketstrukturen wurden über eine alternative Anwendung des dinitzschen Lemmas 5.12 auf reellwertige Kantengewichte anwendbar und haben so zu verbesserten Algorithmen für Graphen mit beliebigen positiven Kantengewichten geführt.

Für die k-kürzeste-Wege-Berechnung haben wir einen für das vergleichsbasierte Modell optimalen Algorithmus angeben können, wenn der k-kürzeste-Wege-Baum explizit berechnet werden soll.

Darüber hinaus gibt die Arbeit einen weit reichenden Überblick über den aktuellen Stand auf dem Gebiet der Kürzeste-Wege-Berechnung – manche der betrachteten Arbeiten sind bislang nur über die Homepages der Autoren erhältlich oder die Algorithmen sind erst in diesem Jahre auf Konferenzen vorstellt worden.

Speziell für Graphen mit negativen Kantengewichten scheint noch Verbesserungspotential vorhanden zu sein: Zum einen reicht die Berechnung einer konsistenten Schätzfunktion in $o(nm)$ bereits aus, um danach auf dem reduzierten Graphen mit positiven reduzierten Kantengewichten die kürzesten Wege effizient zu berechnen und somit den Bellman-Ford-Algorithmus als

schnellsten Algorithmus bei beliebigen Kantengewichten abzulösen. Zum anderen scheint auch eine Verbesserung des Algorithmus für ganzzahlige Kantengewichte möglich. Die Eigenschaften der Graphen in den Teilproblemen wurden noch nicht vollständig genutzt. Die Optimierung der Algorithmen für diese und andere Teilgebiete sollte noch genug Material für zukünftige Forschung bieten.

Anhang A

Heaps und Fibonacci-Heaps

In fast allen in dieser Arbeit vorgestellten Kürzeste-Wege-Algorithmen müssen eine Knotenmenge R und deren Werte $\mathrm{D}(R)$ verwaltet werden. Dazu verwenden wir folgende drei Operationen:

- insert(v, x) fügt den Knoten v mit Wert $\mathrm{D}(v) := x$ in die Menge R ein.
- decrease_key(u, v) verringert für den Knoten v den Wert $\mathrm{D}(v)$ auf $\mathrm{D}(u) + \gamma(uv)$, d.h., $\mathrm{D}(u) + \gamma(uv)$ muss kleiner als der alte Wert $\mathrm{D}(v)$ sein.
- delete_min(R) gibt einen Knoten v mit Wert $\mathrm{D}(v) := \min \mathrm{D}(R)$ zurück und entfernt ihn aus R.

In einem Heap werden dazu zwei Hilfsprozeduren up_heap$(\cdot)$ und down_heap$(\cdot)$ verwendet, die einen Knoten mit Index k in dem mit dem Array assoziierten Binärbaum mit n Elementen aufsteigen bzw. absinken lassen, sodass die Heapbedingung dann wieder hergestellt ist.[1] Die hier nicht näher spezifizierte Prozedur swap(i, j) vertauscht im Array A die Werte an den Positionen i und j. Ferner sei $A[i] := \infty$, $i > n$.

Prozedur A.1 up_heap(k)

$\mathrm{P}_{A.1}.1$ **if** ($k > 1$ and **then** $A[k \text{ div } 2] > A[k]$) **then**
$\mathrm{P}_{A.1}.1\mathrm{a}.1$ swap$(k, k \text{ div } 2)$; up_heap$(k \text{ div } 2)$ **fi**

[1] $A[1]$ ist die Wurzel des Binärbaums (Ebene 1), die Elemente $A[2^{i-1}]$ bis $A[2^i - 1]$ bilden von links nach rechts die Ebene i des Binärbaums. Der Knoten mit Index i hat dann die Söhne mit Index $2i$ und $2i + 1$ und den Vater mit Index i div 2.

Prozedur A.2 down_heap(k, n)

$P_{A.2}.1$ **if** $A[2k] < A[2k+1]$ **then** $h := 2k$ **else** $h := 2k+1$ **fi**;
$P_{A.2}.2$ **if** $A[k] > A[h]$ **then** swap(k, h); down_heap(h, n) **fi**

Die drei oben genannten Prozeduren lassen sich dann leicht implementieren: insert$(\cdot)$ fügt das Element am Ende ein und lässt es mittels up_heap$(\cdot)$ an die richtige Stelle aufsteigen. Der geänderte Knoten nach einem decrease_key$(\cdot)$ steigt ebenso mittels up_heap$(\cdot)$ an die richtige Stelle auf. Das delete_min$(\cdot)$ überschreibt die Wurzel mit dem letzten Element des Heaps und lässt dieses dann mittels down_heap$(\cdot)$ an die richtige Stelle absinken. Diese Operationen benötigen pro Ebene jeweils konstanten Aufwand. Da die Höhe eines vollständigen Binärbaums logarithmisch beschränkt ist, beträgt der Aufwand jeweils $O(\log n)$.

Fibonacci-Heaps (wir orientieren uns an der Darstellung von [Die04]) verbessern diese Schranken auf $O(1)$ für insert$(\cdot)$, decrease_key$(\cdot)$ (und $O(\log n)$ für delete_min$(\cdot)$) im Sinne einer amortisierten Zeitabschätzung, d.h., man betrachtet nicht mehr den Aufwand jeder einzelnen Operation, sondern den Gesamtaufwand einer beliebigen Folge von Operationen und bestimmt dann für jede Operation den Aufwand im Mittel.

Ein Fibonacci-Heap ist eine Liste von knotenbeschrifteten Bäumen, die jeweils die Heapbedingung erfüllen (für alle Kinder j des Knotens i gilt $\mathrm{D}(i) \leq \mathrm{D}(j)$). Da die Anzahl der Kinder sich im Laufe der Operationen verändert, werden die Eltern-Kind-Beziehungen durch Zeiger realisiert und können nicht wie beim oben vorgestellten Heap über einem Feld realisiert werden.

Jeder Knoten (bis auf die Wurzeln der Bäume) kann eine Marke tragen. Ein Knoten ist genau dann markiert, wenn er durch eine decrease_key$(\cdot)$-Operationen schon ein Kind verloren hat. Für jeden Knoten v ist der Rang rank(v) durch die Anzahl seiner Kinder bestimmt. Durch die Markierungen wird gewährleistet, dass die Anzahl der Knoten in einem Baum relativ gross bleibt.[2]

Die Operation insert(v, x) fügt den Knoten v am Ende der Liste an.

Die Operation delete_min$(\cdot)$ ist die einzige, die die Anzahl der Bäume in der Liste verringert.

[2]Wir werden später einen exponentiellen Zusammenhang zwischen der Anzahl der Kinder eines Knotens und der Anzahl der Knoten im gesamten Unterbaum zeigen, sodass für den maximalen Rang $r_{\max}(n)$, der in einem Fibonacci-Heap mit n Elementen auftreten kann, $r_{\max}(n) \in O(\log n)$ gilt.

Funktion A.3 delete_min(R)

$F_{A.3}.1$ $v_{\min} := v$ mit $D(v) = \min\{D(r) \mid r$ ist Wurzel eines Baums$\}$;
$F_{A.3}.2$ Lösche $v_{\min}$; **for all** Kinder v' von $v_{\min}$ **do**
$F_{A.3}.2.1$ hänge den Unterbaum mit Wurzel v' an die Liste an **od**
$F_{A.3}.3$ Erstelle $L[0..r_{\max}(n)]$ mit $L[i] = \{$Liste der Bäume mit Rang $i\}$;
$F_{A.3}.4$ **for** $i := 0$ **to** $r_{\max}(n)$ **do**
$F_{A.3}.4.1$ **while** $|L[i]| \geq 2$ **do**
$F_{A.3}.4.1.1$ Entnehme zwei Bäume aus $L[i]$.
$F_{A.3}.4.1.2$ Hänge den Baum mit dem größeren Wurzel-Schlüsselwert direkt unter die Wurzel des anderen Baumes und füge diesen neuen Baum in $L[i+1]$ an.
$F_{A.3}.4.1$ **od**
$F_{A.3}.4$ **od**;
$F_{A.3}.5$ **return** $v_{\min}$

Nach delete_min($\cdot$) gibt es zu jedem Rang noch höchstens einen Baum, insbesondere ist die Zahl der Bäume durch $r_{\max}(n) + 1$ beschränkt.

Die decrease_key($\cdot$)-Operation verkleinert den Wert $D(\cdot)$ eines Knotens v von $D(v) > D(u)+\gamma(uv)$ auf $D(v) := D(u)+\gamma(uv)$. Dabei werden Unterbäume abgeschnitten und an die Liste der Bäume angehängt sowie Marken eingeführt bzw. wieder entfernt.

Prozedur A.4 decrease_key(u, v)

$P_{A.4}.1$ $D(v) := D(u) + \gamma(uv)$; **if** v ist nicht Wurzel eines Baumes **then**
$P_{A.4}.1b.1$ Sei $vv_1v_2 \dots v_k$ der Weg im Fibonacci-Heap zur Wurzel v_k;
$P_{A.4}.1b.2$ Trenne den Unterbaum mit Wurzel v ab;
$P_{A.4}.1b.3$ Hänge ihn an die Liste der Bäume;
$P_{A.4}.1b.4$ Entferne ggf. die Marke von v;
$P_{A.4}.1b.5$ $i := 1$; **while** v_i hat eine Marke **do**
$P_{A.4}.1b.5.1$ Trenne den Unterbaum mit Wurzel v_i ab;
$P_{A.4}.1b.5.2$ Hänge ihn an die Liste der Bäume;
$P_{A.4}.1b.5.3$ Entferne die Marke von v_i; $i := i + 1$;
$P_{A.4}.1b.5$ **od**;
$P_{A.4}.1b.6$ Markiere v_i, falls v_i keine Wurzel ist
$P_{A.4}.1$ **fi**

Ist v Wurzel, so kann v verringert werden, ohne die Heap-Bedingung zu verletzen, sonst werden auf dem Weg zur Wurzel alle markierten Knoten zu eigenen (unmarkierten) Bäumen – die Schleife bricht spätestens mit $i = k$ ab,

weil v_k als Wurzel nicht markiert ist. Da v nicht unbedingt eine Marke trug, werden also $k-1$ oder k Marken entfernt und 1 oder 0 Marken hinzugefügt. Die Anzahl der Marken sinkt um mindestens $k-2$.[3] Der Aufwand beträgt $k + O(1)$ Schritte. Die Anzahl der Bäume ist um k gestiegen.

Zur Abschätzung der Laufzeiten benötigen wir Aussagen über die Anzahl der Knoten in den Unterbäumen vom Rang i.

Lemma A.5 *Seien v ein Knoten in einem Fibonacci-Heap und c_i der i-te Knoten, der Kind von v geworden ist, dann hat c_i mindestens den Rang $i-2$.*

Hat v den Rang k, $k \geq 0$, so enthält der Unterbaum mit Wurzel k mindestens F_{k+1} Knoten. Hierbei ist F_{k+1} die $(k+1)$-te Fibonacci-Zahl ($F_0 = F_1 = 1$, $F_{k+1} = F_k + F_{k-1}$ für $k \geq 1$).

Beweis: Zu dem Zeitpunkt, als c_i unter den Knoten v gehängt wurde – v war zu diesem Zeitpunkt Wurzel – hatte v bereits die Kinder $c_1, \ldots, c_{i-1}$, der Rang von v war also mindestens $i-1$. Da nur Bäume vom gleichen Rang zu einem Baum vereinigt werden, hatte c_i zu diesem Zeitpunkt ebenso mindestens den Rang $i-1$.

Der Knoten c_i kann inzwischen maximal ein Kind verloren haben (und wäre dann markiert worden – bei Verlust eines weiteren Kindes wäre c_i selbst markiert gewesen und von v abgetrennt worden). Es folgt $\text{rank}(c_i) \geq i-2$.

Sei jetzt B_k die Mindestanzahl der Knoten in einem Unterbaum vom Rang k, $k \geq 0$. Wir zeigen durch Induktion nach k, dass $B_k \geq F_{k+1}$.

Für $k = 0$ und $k = 1$ ist dies wegen $B_0 = 1$ und $B_1 = 2$ korrekt.

Sei jetzt v ein Knoten mit Rang $\text{rank}(v) = k+1$, $k \geq 1$ und die Behauptung richtig für die Ränge $0 \leq i \leq k$.

Seien $c_1, \ldots, c_{k+1}$ die Kinder von v wie oben. Die Knoten c_1 und c_2 können inzwischen ihre Kinder verloren haben. Die anderen c_i haben wie oben gezeigt noch mindestens $i-2$ Kinder. Der Unterbaum von v mit $k+1$ Kindern besteht aus der Wurzel v, dem Kind c_1 (ohne Kinder) und den Kindern $c_2, \ldots, c_{k+1}$, die zusammen mit ihren Unterbäumen nach Induktionsvoraussetzung jeweils

[3] Hatte v keine Marke und ist v_1 nicht die Wurzel, so wurden $k-1=0$ Marken entfernt und 1 Marke hinzugefügt. Die Anzahl der Marken sinkt um $k-2=-1$, steigt also um 1.

mindestens $B_{i-2} \geq F_{i-1}$ Knoten beitragen. Es folgt.

$$B_{k+1} \geq 2 + \sum_{i=2}^{k+1} B_{i-2} \geq 2 + \sum_{i=2}^{k+1} F_{i-1}$$
$$= 1 + \sum_{i=0}^{k} F_i = 1 + \sum_{i=0}^{k} (F_{i+2} - F_{i+1}) = 1 + F_{k+2} - F_1 = F_{k+2}$$

□

Für die Fibonacci-Zahlen gilt $F_n \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$. Somit gilt für den maximalen Rang $r_{\max}(n)$, der in einem Fibonacci-Heap mit n Elementen auftreten kann, $r_{\max}(n) \in O(\log n)$.

Einzelne Fibonacci-Heap-Operationen können deutlich länger dauern als in einem normalen Heap. Wir betrachten hier deshalb eine amortisierte Zeitabschätzung. Die Operationen bauen dabei mit nicht genutzte Zeiteinheiten ein Potential auf, das späteren Operationen zur Verfügung steht. Dieses Potential ist zu Beginn 0 und muss stets positiv bleiben. Der Aufwand der Operationen wird über das Potential $p := T + 2M$ amortisiert, wobei T die Anzahl der Bäume und M die Anzahl der Marken im Fibonacci-Heap angeben. Eine Erhöhung des Potentials muss also mit Zeiteinheiten bezahlt werden, die späteren Operationen dann zur Verfügung stehen.

Satz A.6 *In einer amortisierten Zeitabschätzung benötigen die Operationen in einem Fibonacci-Heap je* $O(1)$ *für* insert$(\cdot)$ *und* decrease_key$(\cdot)$ *sowie* $O(\log n)$ *für* delete_min$(\cdot)$.

Beweis: Das Potential $p := T + 2M$, wobei T die Anzahl der Bäume und M die Anzahl der Marken im Fibonacci-Heap angeben, ist in einem leeren Fibonacci-Heap $p = 0$.

Die Operation insert$(\cdot)$ fügt einen Knoten als neuen Baum in die Liste an – Aufwand $O(1)$, Erhöhung des Potentials $O(1)$, Gesamtaufwand $O(1)$.

Die Operation decrease_key$(\cdot)$ schneidet in k Schritten Unterbäume ab und fügt diese an die Liste an, dabei sinkt die Anzahl der Marken um mindestens $k - 2$ – Aufwand $k + O(1)$, Erhöhung des Potentials um k bzgl. der Anzahl der Bäume, Reduzierung um $2(k - 2)$ bzgl. der Marken, gesamt Potentialveränderung: $-k + O(1)$, Gesamtaufwand $k + O(1) - k + O(1) = O(1)$.

Die Operation delete_min$(\cdot)$ sucht aus T Bäumen die Wurzel mit kleinstem Wert, erhöht die Anzahl der Bäume um höchstens $r_{\max}(n)$, reduziert danach die Anzahl der Bäume auf höchstens $r_{\max}(n) + 1$ – Aufwand $T + O(r_{\max}(n))$,

die Anzahl der Marken bleibt unverändert, das Potential bzgl. der Anzahl der Bäume sinkt um $T - O(r_{\max}(n))$, Gesamtaufwand $O(r_{\max}(n)) \in O(\log n)$. $\square$

Auf den ersten Blick lässt sich das decrease_key$(\cdot)$ auch einfacher implementieren: Man trenne den Unterbaum an dem Knoten, dessen Wert D$(\cdot)$ reduziert wurde, ab und hänge ihn als neuen Baum an. Der Aufwand wäre $O(1)$ und auch das Potential erhöht sich lediglich um 1, sodass a priori auch so der amortisierte Aufwand $O(1)$ beträgt. Dies ist jedoch nicht möglich, da dann die Anzahl der Knoten in den Unterbäumen nicht mehr exponentiell sein müsste (Lemma A.5) und somit $r_{\max}(n)$ und damit auch der Aufwand für delete_min$(\cdot)$ nicht mehr durch $O(\log n)$ abgeschätzt werden könnten.

Literaturverzeichnis

[ACMM93] Azevedo, J.A., M.E.O.S. Costa, J.J.E.R.S. Madeira und E.Q.V. Martins: *An algorithm for the ranking of shortest paths.* European Journal of Operational Research, 69:97–106, 1993.

[AHU75] Aho, Alfred V., John E. Hopcroft und Jeffrey D. Ullman: The *design and analysis of computer algorithms.* Addison-Wesley, Reading, Mass. [u.a.], 2. Auflage, 1975.

[AI00] Asano, Yasuhito und Hiroshi Imai: *Practical Efficiency of the Linear-time Algorithm for the Single Source Shortest Path Problem.* Journal of the Operations Research Society of Japan, 43(4):431–447, December 2000.

[AK94] Abuaiadh, Diab und Jeffrey H. Kingston: *Are Fibonacci Heaps Optimal?* In: Du, Ding-Zhu (Herausgeber): *Algorithms and computation : 5th International Symposium, Beijing, P.R. China, August 25 - 27, 1994 ; proceedings / ISAAC '94*, Band 834 der Reihe *Lecture Notes in Computer Science*, Seiten 442–450, 1994.

[AMMP94] Azevedo, J.A., J.J.E.R.S. Madeira, E.Q.V. Martins und F.M.A. Pires: *A computational improvement for a shortest paths ranking algorithm.* European Journal of Operational Research, 73:188–191, 1994.

[AMO93] Ahuja, Ravindra K., Thomas L. Magnanti und James B. Orlin: *Network Flows : Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[AMOT90] Ahuja, Ravindra K., Kurt Mehlhorn, James Orlin und Robert E. Tarjan: *Faster algorithms for the shortest path problem.* Journal of the ACM, 37(2):213–223, 1990.

[AMT99] Andersson, Arne, Peter Bro Miltersen und Mikkel Thorup: *Fusion trees can be implemented with AC^0 instructions only.* Theoretical Computer Science, 215:337–344, 1999.

[BCKM01] Brodnik, Andrej, Svante Carlsson, Johan Karlsson und J. Ian Munro: *Worst case constant time priority queue.* In: *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, Seiten 523–528. Society for Industrial and Applied Mathematics, 2001.

[Bel58] Bellman, Richard: *On a Routing Problem.* Quarterly of Applied Mathematics, 16(1):87–90, 1958.

[BM83] Bagchi, A. und A. Mahanti: *Search Algorithms Under Different Kinds of Heuristics – A Comparative Study.* Journal of the ACM, 30(1):1–21, 1983.

[Bod93] Bodlaender, Hans J.: *A Tourist Guide through Treewidth.* Acta Cybernetica, 11:1–22, 1993. http://citeseer.ist.psu.edu/bodlaender93tourist.html.

[Bod97] Bodlaender, Hans J.: *Treewidth: Algorithmic Techniques and Results.* In: *Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97, Bratislava, Slovakia, August 25-29, 1997, Proceedings*, Band 1295 der Reihe *Lecture Notes in Computer Science*, Seiten 19–36, 1997. http://citeseer.ist.psu.edu/bodlaender98treewidth.html.

[Bra94] Brandstädt, Andreas: *Graphen und Algorithmen.* Leitfäden und Monographien der Informatik. Teubner, Stuttgart, 1994.

[BS81] Bronstein, Ilja N. und Konstantin A. Semendjajew: *Taschenbuch der Mathematik.* H. Deutsch, Thun ; Frankfurt, 20. Auflage, 1981.

[Buc00] Buchholz, Friedhelm: *Hierarchische Graphen zur Wegesuche.* Dissertation, Institut für Informatik, Universität Stuttgart, 2000.

[CFMP00] Cooper, Colin, Alan Frieze, Kurt Mehlhorn und Volker Priebe: *Average-case complexity of shortest-paths problems in the vertex-potential model.* Random Structures and Algorithms, 16(1):33–46, 2000.

[CGR96] Cherkassky, Boris V., Andrew V. Goldberg und Tomasz Radzik: *Shortest Paths Algorithms: Theory and Experimental Evaluation.* Mathematical Programming, 73:129–174, 1996. auch in: SODA '94, 516–525. http://citeseer.ist.psu.edu/cherkassky93shortest.html. Source Code: http://www.avglab.com/andrew/soft/splib.tar.

[CGS99] Cherkassky, Boris V., Andrew V. Goldberg und Craig Silverstein: *Buckets, Heaps, Lists, and Monotone Priority Queues.* SIAM Journal on Computing, 28(4):1326–1346, 1999.

[CLRS01] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms.* MIT Press, Cambridge, Mass. [u.a.], 2. Auflage, 2001.

[CM99] Chiang, Yi-Jen und Joseph S. B. Mitchell: *Two-point Euclidean shortest path queries in the plane.* In: *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, Seiten 215–224. Society for Industrial and Applied Mathematics, 1999.

[CR73] Cook, Stephen A. und Robert A. Reckhow: *Time Bounded Random Access Machines.* Journal of Computer and System Sciences, 7:354–375, 1973.

[CW79] Carter, J. Lawrence und Mark N. Wegman: *Universal Classes of Hash Functions.* Journal of Computer and System Sciences, 18(2):143–154, 1979.

[CW90] Coppersmith, Don und Shmuel Winograd: *Matrix multiplication via arithmetic progressions.* Journal of Symbolic Computation, 9(3):251–280, 1990.

[CZ98] Chaudhuri, Shiva und Christos D. Zaroliagis: *Shortest Paths in Digraphs of Small Treewidth. Part II: Optimal Parallel Algorithms.* Theoretical Computer Science, 203(2):205–223, 1998. http://citeseer.ist.psu.edu/chaudhuri97shortest.html.

[CZ00] Chaudhuri, Shiva und Christos D. Zaroliagis: *Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms.* Algorithmica, 27(3):212–226, 2000. http://citeseer.ist.psu.edu/260522.html.

[dC83] Champeaux, Dennis de: *Bidirectional Heuristic Search Again.* Journal of the ACM, 30(1):22–32, 1983.

[dCS77] Champeaux, Dennis de und Lenie Sint: *An Improved Bidirectional Heuristic Search Algorithm.* Journal of the ACM, 24(2):177–191, 1977.

[DF79] Denardo, Eric V. und Bennett L. Fox: *Shortest-Route Methods: 1. Reaching, Pruning, and Buckets.* Operations Research, 27(1):161–186, 1979.

[DHKP97] Dietzfelbinger, Martin, Torben Hagerup, Jyrki Katajainen und Martti Penttonen: *A Reliable Randomized Algorithm for the Closest-Pair Problem.* Journal of Algorithms, 25(1):19–51, 1997.

[Dia69] Dial, Robert B.: *Algorithm 360: shortest-path forest with topological ordering [H].* Communications of the ACM, 12(11):632–633, 1969.

[Die04] Diekert, Volker: *Skript zur Vorlesung: Entwurf und Analyse effizienter Algorithmen.* Universität Stuttgart, Sommersemester 2004.

[Dij59] Dijkstra, E. W.: *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik, 1(4):269–271, 1959.

[Din78] Dinic, E. A.: *Economical Algorithms for Finding Shortest Paths in a Network.* In: Popkov, Y. und S. Shmulyian (Herausgeber): *Transportation Modeling Systems*, Seiten 36–44, 1978.

[Dji82] Djidjev, H.N.: *On the problem of partitioning planar graphs.* SIAM Journal on Algebraic and Discrete Methods, 3:229–240, 1982.

[DP84] Deo, Narsingh und Chi-Yin Pang: *Shortest-Path Algorithms: Taxonomy and Annotation.* Networks, 14:275–323, 1984.

[DP85] Dechter, Rina und Judea Pearl: *Generalized best-first search strategies and the optimality af A*.* Journal of the ACM, 32(3):505–536, 1985.

[DPZ00] Djidjev, H. N., G. E. Pantziou und C. D. Zaroliagis: *Improved Algorithms for Dynamic Shortest Paths.* Algorithmica, 28:367–389, 2000.

[Dre69] Dreyfus, Stuart E.: *An Appraisal of Some Shortest-Path Algorithms.* Operations Research, 17(3):395–412, 1969.

[DV97] Djidjev, H.N. und S.M. Venkatesan: *Reduced constants for simple cycle graph separation.* Acta Informatica, 34:231–243, 1997.

[EK72] Edmonds, Jack und Richard M. Karp: *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems.* Journal of the ACM, 19(2):248–264, 1972.

[EP04] Elkin, Michael und David Peleg: $(1 + \epsilon, \beta)$-*Spanner Constructions for General Graphs.* SIAM Journal on Computing, 33(3):608–631, 2004.

[Epp98] Eppstein, David: *Finding the k Shortest Paths.* SIAM Journal on Computing, 28(2):652–673, 1998.

[Epp01] Eppstein, David: *Bibliography on algorithms for k shortest paths.* http://liinwww.ira.uka.de/bibliography/Theory/k-path.html, 2001.

[FF62] Ford, Lester R. und Delbert R. Fulkerson: *Flows in networks.* Princeton University Press, Princeton, N. J., 1962.

[Flo62] Floyd, Robert W.: *Algorithm 97: Shortest path.* Communications of the ACM, 5(6):345, 1962.

[For56] Ford, L. R.: *Network flow theory.* Technical Report P-923, The Rand Corporation, Santa Monica, CA, August 1956.

[Fox73] Fox, B.L.: *Calculating kth Shortest Paths.* INFOR, 11(1):66–70, 1973.

[Fre76] Fredman, M.L.: *New Bounds on the Complexity of the Shortest Path Problem.* SIAM Journal on Computing, 5(1):83–89, 1976.

[Fre87] Frederickson, G.N.: *Fast Algorithms for Shortest Paths in Planar Graphs, with Applications.* SIAM Journal on Computing, 16:1004–1022, 1987.

[Fre93] Frederickson, Greg N.: *An Optimal Algorithm for Selection in a Min-Heap.* Information and Computation, 104:197–214, 1993.

[FT87] Fredman, Michael L. und Robert Endre Tarjan: *Fibonacci heaps and their uses in improved network optimization algorithms*. Journal of the ACM, 34(3):596–615, 1987.

[FW94] Fredman, Michael L. und Dan E. Willard: *Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths*. Journal of Computer and System Sciences, 48:533–551, 1994.

[Gab85] Gabow, Harold N.: *Scaling algorithms for network problems*. Journal of Computer and System Sciences, 31(2):148–168, 1985.

[Gel77] Gelperin, David: *On the Optimality of A^**. Artificial Intelligence, 8(1):69–76, 1977.

[GH04] Goldberg, A. V. und C. Harrelson: *Computing the Shortest Path: A* Search Meets Graph Theory*. Technical Report MSR-TR-2004-24, Microsoft Research, 2004. http://www.avglab.com/andrew/pub/msr-tr-2004-24.ps.

[GJ79] Garey, Michael R. und David S. Johnson: *Computer and Intractability : a Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[GKP85] Glover, F., D. Klingman und N. Phillips: *A New Polynomially Bounded Shortest Path Algorithm*. Operations Research, 33(1):65–73, 1985.

[Gol95] Goldberg, Andrew V.: *Scaling Algorithms for the Shortest Paths Problem*. SIAM Journal on Computing, 24(3):494–504, June 1995. auch in: Symposium on Discrete Algorithms SODA'93, 222–231.

[Gol01a] Goldberg, Andrew V.: *Shortest Path Algorithms: Engineering Aspects*. In: *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC '01)*, Band 2223 der Reihe *Lecture Notes in Computer Science*, Seiten 502–513, 2001. http://www.avglab.com/andrew/soft.html.

[Gol01b] Goldberg, Andrew V.: *A Simple Shortest Path Algorithm with Linear Average Time*. In: *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, Band 2161 der Reihe *Lecture Notes in Computer Science*, Seiten 230–241, 2001. http://www.avglab.com/andrew/soft.html.

[Gol04] Goldberg, Andrew V.: *A Practical Shortest Path Algorithm with Linear Expected Time.* SIAM Journal on Computing, submitted: , 2004. http://www.avglab.com/andrew/pub/sp-lin.ps.

[GP86] Gallo, Giorgio und Stefano Pallottino: *Shortest Path Methods: A Unifying Approach.* Mathematical Programming Study, 26:38–64, 1986.

[GP88] Gallo, Giorgio und Stefano Pallottino: *Shortest Path Algorithms.* Annals of Operations Research, 13:3–79, 1988.

[GR93] Goldberg und Radzik: *A Heuristic Improvement of the Bellman-Ford Algorithm.* Applied Mathematics Letters, 6(3):3–6, 1993.

[GT89] Gabow, Harold N. und Robert E. Tarjan: *Faster Scaling Algorithms for Network Problems.* SIAM Journal on Computing, 18(5):1013–1036, October 1989.

[Hag98] Hagerup, Torben: *Sorting and Searching on the Word RAM.* In: Morvan, Michel, Christoph Meinel und Daniel Krob (Herausgeber): *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998, Proceedings*, Band 1373 der Reihe *Lecture Notes in Computer Science*, Seiten 366–398. Springer-Verlag, 1998.

[Hag00] Hagerup, Torben: *Improved Shortest Paths on the Word RAM.* In: Montanari, Ugo, José D.P. Rolim und Emo Welzl (Herausgeber): *Automata, Languages and Programming: 27th International Colloquium ; Proceedings / ICALP 2000, Geneva, Switzerland, July 9 - 15, 2000.*, Band 1853 der Reihe *Lecture Notes in Computer Science*, Seiten 61–72. Springer, 2000.

[Hag04] Hagerup, Torben: *Simpler Computation of Single-Source Shortest Paths in Linear Average Time.* In: Diekert, Volker und Michel Habib (Herausgeber): *STACS 2004, 21st Annual Symposium on Theoretical Aspects of Computer Science, Montpellier, France, March 25-27, 2004, Proceedings*, Band 2996 der Reihe *Lecture Notes in Computer Science*, Seiten 362–369, 2004.

[HKRS97] Henzinger, Monika, Philip Klein, Satish Rao und Sairam Subramanian: *Faster Shortest-Path Algorithms*

for Planar Graphs. Journal of Computer and System Sciences, 55:3–23, 1997. auch: 26th STOC '94, http://citeseer.ist.psu.edu/henzinger94faster.html.

[HMS03] Hershberger, John E., Matthew Maxel und Subhash Suri: *Finding the k Shortest Simple Paths: A New Algorithm and its Implementation.* In: *Proceedings of the 5th Workshop on Algorithm Engineering & Experiments (ALENEX)*, 2003. http://www.siam.org/meetings/alenex03/Abstracts/ jhershberger.pdf.

[HNR68] Hart, P., N. Nilsson und B. Raphael: *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* IEEE Transactions on Systems Science and Cybernetics, SSC-4(2):100–107, 1968.

[Hol03] Holzer, Martin: *Hierarchical Speed-up Techniques for Shortest-Path Algorithms.* Diplomarbeit, Dept. of Informatics, University of Konstanz, Germany, February 2003. http://www.ub.uni-konstanz.de/kops/volltexte/2003/1038/.

[HP59] Hoffman, Walter und Richard Pavley: *A Method for the Solution of the Nth Best Path Problem.* Journal of the ACM, 6(4):506–514, 1959.

[HS99] Hershberger, John und Subhash Suri: *An Optimal Algorithm for Euclidean Shortest Paths in the Plane.* SIAM Journal on Computing, 28(6):2215–2256, 1999.

[HZ85] Hassin, Refael und Eitan Zemel: *On Shortest Paths in Graphs with Random Weights.* Mathematics of Operations Research, 10(4):557–564, 1985.

[IHI94] Ikeda, Takahiro, Min-Yao Hsu und Hiroshi Imai: *A Fast Algorithm for Finding Better Routes by AI Search Techniques.* In: *1994 Vehicle Navigation & Information Systems Conference Proceedings*, Seiten 291–296, 1994.

[Joh73] Johnson, Donald B.: *A Note on Dijkstra's Shortest Path Algorithm.* Journal of the ACM, 20(3):385–388, 1973.

[Joh77] Johnson, Donald B.: *Efficient Algorithms for Shortest Paths in Sparse Networks.* Journal of the ACM, 24(1):1–13, 1977.

[Joh02] Johnson, David. S.: *A Theoretician's Guide to the Experimental Analysis of Algorithms.* In: Goldwasser, M. H., D. S. Johnson und C. C. McGeoch (Herausgeber): *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, Seiten 215–250. American Mathematical Society, Providence, 2002. http://www.research.att.com/~dsj/papers.html.

[Ker81] Kershenbaum, Aaron: *A Note on Finding Shortest Path Trees.* Networks, 11:399–400, 1981.

[KIM82] Katoh, N., T. Ibaraki und H. Mine: *An Efficient Algorithm for K Shortest Simple Paths.* Networks, 12:411–427, 1982.

[KK97] Kaindl, Hermann und Gerhard Kainz: *Bidirectional Heuristic Search Reconsidered.* Journal of Artificial Intelligence Research, 7:283–317, 1997.

[KKP93] Karger, David R., Daphne Koller und Steven J. Phillips: *Finding the Hidden Path: Time Bounds for All-Pairs Shortest Paths.* SIAM Journal on Computing, 22(6):1199–1217, 1993. auch in: FOCS '91, 560–568.

[Kle02] Klein, Philip: *Preprocessing an undirected planar network to enable fast approximate distance queries.* In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, Seiten 820–827. Society for Industrial and Applied Mathematics, 2002.

[Kle04] Klein, Philip N.: *Multiple-source shortest paths in planar graphs allowing negative lengths.* Brown University, 2004. http://www.cs.brown.edu/people/klein/publications/2004mssp.ps.

[Koz92] Kozen, Dexter: The *Design and Analysis of Algorithms.* Texts and monographs in computer science. Springer, New York ; Berlin ; Heidelberg [u.a.], 1992.

[Kwa89] Kwa, James B.H.: *BS*: An Admissable Bidirectional Staged Heuristic Search Algorithm.* Artificial Intelligence, 38(1):95–109, 1989.

[Law72] Lawler, Eugene L.: *A Procedure for Computing the K Best Solutions to Discrete Optimization Problems and Its Application*

to the Shortest Path Problem. Management Science, 18(7):401–405, 1972.

[Lew97] Lewandowski, Stefan: *Anwendung für Separatortheoreme auf planaren Graphen.* Diplomarbeit Nr. 1508, Fakultät Informatik, Universität Stuttgart, Stuttgart, 1997.

[LR89] Luby, Michael und Prabhakar Ragde: *A Bidirectional Shortest-Path Algorithm with Good Average-Case Behavior.* Algorithmica, 4:551–567, 1989.

[LT79] Lipton, R.J. und R.E. Tarjan: *A Separator Theorem for Planar Graphs.* SIAM Journal on Applied Mathematics, 36:177–189, 1979.

[LT80] Lipton, R.J. und R.E. Tarjan: *Applications of a Planar Separator Theorem.* SIAM Journal on Computing, 9:615–627, 1980.

[Mar77] Martelli, Alberto: *On the Complexity of Admissible Search Algorithms.* Artificial Intelligence, 8:1–13, 1977.

[McG95] McGeoch, Catherine C.: *All-Pairs Shortest Paths and the Essential Subgraph.* Algorithmica, 13(5):426–441, 1995.

[Meh84] Mehlhorn, Kurt: *Data structures and algorithms.* EATCS monographs on theoretical computer science. Springer, Berlin [u.a.], 1984.

[Mér81] Mérõ, László: *Some Remarks on Heuristic Search Algorithms.* In: *Proceedings of the Seventh International Joint Conference on Artificial Intelligence : IJCAI-81 ; 24 - 28 August 1981, University of British Columbia, Vancouver, B. C., Canada*, Band 1. Kaufmann, 1981.

[Mey01] Meyer, Ulrich: *Single-source shortest-paths on arbitrary directed graphs in linear average-case time.* In: *Symposium on Discrete Algorithms*, Seiten 797–806, 2001.

[Mey02] Meyer, Ulrich: *Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms.* Dissertation, Universität des Saarlandes, 2002. http://www.mpi-sb.mpg.de/~umeyer/pubs.html.

[Mey03] Meyer, Ulrich: *Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds*. Journal of Algorithms, 48(1):91–134, August 2003.

[Mit00] Mitchell, J.S.B.: *Geometric Shortest Paths and Network Optimization*. In: Sack, J.-R. und J. Urrutia (Herausgeber): *Handbook of Computational Geometry*, Seiten 633–701. Elsevier Science, 2000. http://www.ams.stonybrook.edu/~jsbm/publications.html.

[MN99] Mehlhorn, K. und St. Näher: *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html.

[MPS99] Martins, E.Q.V., M.M.B. Pascoal und J.L.E. Santos: *Deviation Algorithms for Ranking Shortest Paths*. International Journal of Foundations of Computer Science, 10(3):247–261, 1999.

[MPSS02] Mehlhorn, Kurt, Volker Priebe, Guido Schäfer und Naveen Sivadasan: *All-pairs shortest-paths computation in the presence of negative cycles*. Information Processing Letters, 81(6):341–343, March 2002. auch: http://www.mpi-sb.mpg.de/~schaefer/ftp/ps/APSP-ipl.ps.gz.

[MS03] Meyer, U. und P. Sanders: Δ-*stepping: a parallelizable shortest path algorithm*. Journal of Algorithms, 49(1):114–152, October 2003.

[MT87] Moffat, Alistair und Tadao Takaoka: *An All Pairs Shortest Path Algorithm with Expected Time* $O(n^2 \log n)$. SIAM Journal on Computing, 16(6):1023–1031, 1987.

[Nic66] Nicholson, T.A.J.: *Finding the shortest route between two points in a network*. Computer Journal, 9(3):275–280, November 1966.

[OA92] Orlin, James B. und Ravindra K. Ahuja: *New Scaling Algorithms for the Assignment and Minimum Mean Cycle Problems*. Mathematical Programming, 54:41–56, 1992.

[Pal84] Pallottino, Stefano: *Shortest-Path Methods: Complexity, Interrelations and New Propositions.* Networks, 14:257–267, 1984.

[Pap74] Pape, U.: *Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem.* Mathematical Programming, 7:212–222, 1974.

[Pap80] Pape, U.: *Algorithm 562: Shortest Path Lengths [H].* ACM Transactions on Mathematical Software, 6(3):450–455, 1980.

[Pap83] Pape, U.: *Remark on algorithm 562: shortest path lengths.* ACM Transactions on Mathematical Software, 9(2):260, 1983.

[Pea84] Pearl, Judea: *Heuristics : intelligent search strategies for computer problem solving.* The Addison-Wesley series in artificial intelligence. Addison-Wesley, Reading, Mass., 1984.

[Pet02] Pettie, S.: *A faster all-pairs shortest path algorithm for real-weighted sparse graphs.* Technischer Bericht CS-TR-02-13, UT-CS, February 2002.

[Pet03] Pettie, Seth: *On the Shortest Path and Minimum Spanning Tree Problems.* Doktorarbeit, University of Texas at Austin, 2003. http://www.mpi-sb.mpg.de/~pettie/.

[Pet04] Pettie, S.: *A New Approach to All-Pairs Shortest Paths on Real-Weighted Graphs.* Theoretical Computer Science, 312(1):47–74, January 2004. auch ICALP'02: A Faster All-pairs Shortest Path Algorithm for Real-weighted Sparse Graphs.

[Poh71] Pohl, Ira: *Bi-Directional Search.* In: Meltzer, Bernard (Herausgeber): *Machine intelligence 6*, Seiten 127–140, Edinburgh, 1971. University Press.

[PP84] Politowski, George und Ira Pohl: *D-Node Retargeting in Bidirectional Heuristic Search.* In: *Proceedings of the National Conference on Artificial Intelligence*, Seiten 274–277, 1984.

[PR02] Pettie, S. und V. Ramachandran: *Computing shortest paths with comparisons and additions.* In: *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, Seiten 267–276, 2002. Aktualisierte erweiterte Fassung: http://www.mpi-sb.mpg.de/~pettie/.

[Pri01] Priebe, Volker: *Average-Case Complexity of Shortest-Paths Problems.* Dissertation, Universität des Saarlandes, 2001. http://www.mpi-sb.mpg.de/~priebe/.

[PRS02] Pettie, S., V. Ramachandran und S. Sridhar: *Experimental evaluation of a new shortest path algorithm.* In: *4th Workshop on Algorithm Engineering and Experiments (ALENEX'02), LNCS Vol. 2409*, Seiten 126–142, 2002.

[PS97] Pallottino, Stefano und Maria Grazia Scutella': *Shortest Path Algorithms in Transportation models: classical and innovative aspects.* Technischer Bericht TR-97-06, 14, 1997. http://citeseer.ist.psu.edu/pallottino98shortest.html.

[PSWZ04] Pyrga, Evangelia, Frank Schulz, Dorothea Wagner und Christos Zaroliagis: *Experimental Comparison of Shortest Path Approaches for Timetable Information.* In: *Proceedings 5th Workshop on Algorithm Engineering and Experiments*, 2004. http://www.siam.org/meetings/alenex04/abstacts/epyrga.pdf.

[PW60] Pollack, Maurice und Walter Wiebenson: *Solutions of the Shortest-Route Problem-A Review.* Operations Research, 8(2):224–230, 1960.

[Ram97] Raman, Rajeev: *Recent results on the single-source shortest paths problem.* SIGACT News, 28(2):81–87, 1997.

[Sch00a] Schmid, Wolfgang: *Berechnung kürzester Wege in Straßennetzen mit Wegeverboten.* Dissertation, Fakultät für Bauingenieur- und Vermessungswesen, Universität Stuttgart, 2000.

[Sch00b] Schulz, Frank: *Effiziente Algorithmen für ein Fahrplanauskunftssystem.* Konstanzer Schriften in Mathematik und Informatik Nr. 110, Fakultät für Mathematik und Informatik, Universität Konstanz, Januar 2000. http://www.fmi.uni-konstanz.de/Schriften.

[Shi76] Shier, D.R.: *Iterative Methods for Determining the k Shortest Paths in a Network.* Networks, 6:205–229, 1976.

[SII+95] Shibuya, Tetsuo, T. Ikeda, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura und Kenji Tenmoku: *Finding a realistic detour by AI search techniques.* In: *Proc. 2nd Intelligent*

Transportation Systems, Band 4, Seiten 2037–2044, November 1995.

[ST03] Saunders, Shane und Tadao Takaoka: *Improved shortest path algorithms for nearly acyclic graphs*. Theoretical Computer Science, 293:535–556, 2003.

[Str69] Strassen, Volker: *Gaussian Elimination is not Optimal*. Numerische Mathematik, 13, 1969.

[SV86] Sedgewick, Robert und Jeffrey Scott Vitter: *Shortest Paths in Euclidean Graphs*. Algorithmica, 1:31–48, 1986.

[SW81] Shier, Douglas R. und Christoph Witzgall: *Properties of Labeling Methods for Determining Shortest Path Trees*. Journal of Research of the National Bureau of Standards, 86(3):317–330, May-June 1981.

[SWW00] Schulz, Frank, Dorothea Wagner und Karsten Weihe: *Dijkstra's algorithm on-line: an empirical case study from public railroad transport*. Journal of Experimental Algorithmics, 5:12, 2000. http://www.jea.acm.org/.

[SWZ02] Schulz, Frank, Dorothea Wagner und Christos Zaroliagis: *Using Multi-Level Graphs for Timetable Information*. In: *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX 2002)*, Band 2409 der Reihe *Lecture Notes in Computer Science*, Seiten 43–59. Springer, 2002.

[SZ99] Shoshan, Avi und Uri Zwick: *All Pairs Shortest Paths in Undirected Graphs with Integer Weights*. In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*, Seiten 605–615, 1999.

[Tak92] Takaoka, Tadao: *A new upper bound on the complexity of the all pairs shortest path problem*. Inf. Process. Lett., 43(4):195–199, 1992.

[Tak98a] Takaoka, Tadao: *Shortest path algorithms for nearly acyclic directed graphs*. Theoretical Computer Science, 203:143–150, 1998. auch in: Workshop on Graph-Theoretic Concepts in Computer Science, 1996. http://citeseer.ist.psu.edu/takaoka97shortest.html.

[Tak98b] Takaoka, Tadao: *Subcubic Cost Algorithms for the All Pairs Shortest Path Problem.* Algorithmica, 20:309–318, 1998.

[Tar72] Tarjan, Robert: *Depth-First Search and Linear Graph Algorithms.* SIAM Journal on Computing, 1(2):146–160, 1972.

[Tar83] Tarjan, Robert E.: *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, 1983.

[Tho99] Thorup, Mikkel: *Undirected single-source shortest paths with positive integer weights in linear time.* Journal of the ACM, 46(3):362–394, 1999.

[Tho00] Thorup, Mikkel: *On RAM priority queues.* SIAM Journal on Computing, 30:86–109, 2000. auch Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms '96, 59–67.

[Tho01] Thorup, Mikkel: *Compact Oracles for Reachability and Approximate Distances in Planar Digraphs.* In: *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS'01)*, Seiten 242–251, 2001.

[Tho03] Thorup, Mikkel: *Integer priority queues with decrease key in constant time and the single source shortest paths problem.* In: *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, Seiten 149–158. ACM Press, 2003.

[Tur96] Turau, Volker: *Algorithmische Graphentheorie.* Addison-Wesley, 1996.

[TZ01] Thorup, Mikkel und Uri Zwick: *Approximate distance oracles.* In: *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, Seiten 183–192. ACM Press, 2001.

[WW03] Wagner, Dorothea und Thomas Willhalm: *Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs.* Konstanzer Schriften in Mathematik und Informatik Nr. 183, Universität Konstanz, Januar 2003. http://www.fmi.uni-konstanz.de/Schriften.

[Yen71] Yen, Jin Y.: *Finding the K Shortest Loopless Paths in a Network.* Management Science, 17(11):712–716, July 1971.

[ZN98] Zhan, F. Benjamin und Charles E. Noon: *Shortest Path Algorithms: An Evaluation using Real Road Networks.* Transportation Science, 32(1):65–73, 1998. http://www.swt.edu/~fz01/Pubs.html.

[ZN00] Zhan, F. Benjamin und Charles E. Noon: *A Comparison Between Label-Setting and Label-Correcting Algorithms for Computing One-to-One Shortest Paths.* Journal of Geographic Information and Decision Analysis, 4(2):1–11, 2000. http://www.swt.edu/~fz01/Pubs.html.

[Zwi01] Zwick, Uri: *Exact and Approximate Distances in Graphs - A Survey.* In: *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, Band 2161 der Reihe *Lecture Notes in Computer Science*, Seiten 33–48. Springer, 2001. Aktualisierte Fassung: http://www.cs.tau.ac.il/~zwick/.

[Zwi02] Zwick, Uri: *All pairs shortest paths using bridging sets and rectangular matrix multiplication.* Journal of the ACM, 49(3):289–317, 2002.

Lebenslauf

Geburtstag:	5. Juli 1972
Geburtsort:	Northeim
Familienstand:	verheiratet

1978–1982:	Martin-Luther-Schule, Northeim
1982–1984:	Thomas-Mann-Schule, Northeim
1984–1991:	Gymnasium Corvinianum, Northeim

1991–1992:	Zivildienst im Altenheim der Inneren Mission e.V., Northeim

1992–1995:	Studium der Informatik an der Philipps-Universität Marburg
1995–1997:	Studium der Informatik an der Universität Stuttgart

1997–2004:	Wissenschaftlicher Angestellter bei Prof. Dr. V. Claus, Abteilung Formale Konzepte, Institut für Formale Methoden der Informatik an der Universität Stuttgart
seit 1.11.2004:	Wissenschaftlicher Assistent bei Prof. Dr. V. Claus, Abteilung Formale Konzepte, Institut für Formale Methoden der Informatik an der Universität Stuttgart