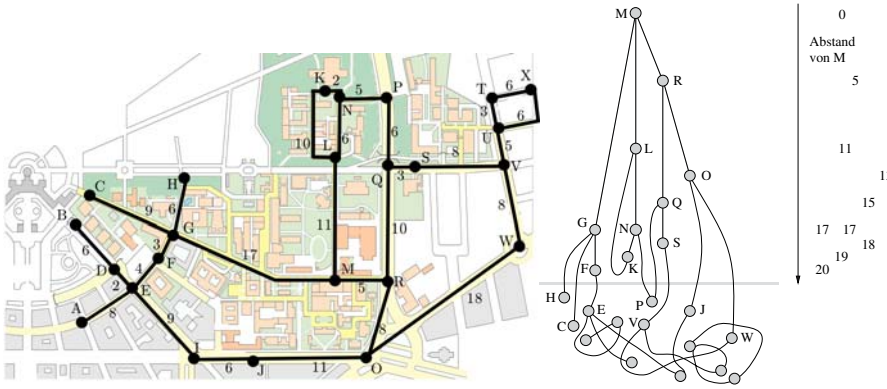
10

Kürzeste Wege



Das Problem, in einem Netzwerk den kürzesten, schnellsten oder billigsten Weg zu finden, ist allgegenwärtig. Jeder von uns löst es jeden Tag. Wenn man an Ort s ist und zu einem Ort t gelangen möchte, fragt man nach dem Weg, der einen am schnellsten von s nach t bringt. Die Feuerwehr möchte vielleicht die schnellsten Wege von der Feuerwehrzentrale s zu allen Stellen t in der Stadt finden – dies führt zum Problem „mit einem Startpunkt" (engl.: single source*). Manchmal möchte man auch eine vollständige Tabelle aller Distanzen von jedem Ort zu jedem anderen haben – dies ist das Problem „kürzeste Wege für alle (Knoten-)Paare" (engl.:* all pairs*). Zum Beispiel findet sich in einem Straßenatlas üblicherweise eine Tabelle mit den Distanzen zwischen allen größeren Städten.*

Es gibt einen Routenplanungsalgorithmus, der nur einen Stadtplan und eine Menge Geschicklichkeit, aber keinen Computer benötigt. Man legt dünne Fäden entlang der Straßen auf dem Stadtplan. An Kreuzungen werden die Fäden verknotet, und auch für den Startpunkt gibt es einen Knoten. Nun hebt man den Startknoten langsam hoch, bis das gesamte Netz unter ihm hängt. Wenn man dies schafft, ohne dass sich die Fäden verheddern, und wenn die Knoten so klein sind, dass nur ein straff gespannter Faden einen Knoten daran hindert, nach unten zu fallen, dann definieren die straffen Fäden genau die kürzesten Wege. Das einführende Bild[1] *für dieses Kapitel zeigt eine Karte des Campus Süd des Karlsruher Instituts für Technologie (KIT) und illustriert den „fadenbasierten" Routenplanungsalgorithmus für den Startknoten M.*

Routenplanung in Straßennetzen ist eine der vielen Anwendungen von Kürzeste-Wege-Berechnungen. Aber auch viele andere Probleme lassen sich mit Hilfe von Kürzeste-Wege-Berechnungen besser lösen, wenn man eine passende Modellierung

[1] © KIT, Institut für Photogrammetrie und Fernerkundung.

M. Dietzfelbinger et al., *Algorithmen und Datenstrukturen,* eXamen.press,
DOI 10.1007/978-3-642-05472-3_10, © Springer-Verlag Berlin Heidelberg 2014

über Graphen findet. Beispielsweise werden in Ahuja *et al.* [8] so verschiedenartige Anwendungen genannt wie Planung von Flüssen in Netzwerken, Planung des Wohnungsbaus in Städten, Planung des Lagerbestands in Produktion und Handel, DNA-Sequenzierung, das Rucksackproblem (siehe auch Kap. 12), Produktionsplanung, Erstellung von Dienstplänen z. B. für Callcenter, Wagenparkplanung, die Approximation von stückweise linearen Funktionen und die Planung von Inspektionen bei einer Fertigungsstraße.

In der allgemeinsten Formulierung des Kürzeste-Wege-Problems betrachtet man einen gerichteten Graphen $G = (V, E)$ und eine Kostenfunktion c, die jeder Kante einen beliebigen reellen Kostenwert zuordnet. Es stellt sich heraus, dass es beträchtlichen Zeitaufwand erfordert, das Problem in dieser allgemeinen Formulierung zu lösen. Deshalb interessieren wir uns auch für verschiedene Einschränkungen, die einfachere und effizientere Algorithmen ermöglichen: nichtnegative Kantenkosten, ganzzahlige Kantenkosten und azyklische Graphen. Man beachte, dass der sehr spezielle Fall von Kanten mit Kosten 1 schon in Kap. 9.1 gelöst wurde – der Breitensuchbaum mit Wurzel s ist eine kompakte Darstellung von kürzesten Wegen von s aus zu allen anderen Knoten. In Abschnitt 10.1 betrachten wir einführend einige grundlegende Begriffe, die zu einem allgemeinen Ansatz für Kürzeste-Wege-Algorithmen führen. Ein systematisches Vorgehen wird uns helfen, in der Vielzahl verschiedener Kürzeste-Wege-Algorithmen den Überblick zu behalten. Als erstes Beispiel für einen schnellen und einfachen Algorithmus für eine eingeschränkte Klasse von Eingaben betrachten wir in Abschnitt 10.2 ein Verfahren für azyklische Graphen. In Abschnitt 10.3 kommen wir zum meistbenutzten Algorithmus für kürzeste Wege, nämlich dem Algorithmus von Dijkstra für allgemeine gerichtete Graphen mit nichtnegativen Kantenkosten. Wie effizient der Algorithmus von Dijkstra wirklich ist, hängt davon ab, wie effizient die verwendeten Prioritätswarteschlangen sind. In einer einführenden Vorlesung oder beim ersten Lesen kann man nach dem Algorithmus von Dijkstra vielleicht mit dem Thema „Kürzeste Wege“ aufhören. Aber im restlichen Teil des Kapitels werden noch viele andere interessante Dinge zu diesem Thema präsentiert. Wir beginnen mit einer Durchschnittsanalyse des Algorithmus von Dijkstra in Abschnitt 10.4, die darauf hindeutet, dass im Normalfall die Operationen der Prioritätswarteschlange die Rechenzeit weniger dominieren als man nach der Analyse glauben könnte. In Abschnitt 10.5 diskutieren wir *monotone Prioritätswarteschlangen für ganzzahlige Schlüssel*, die eine spezielle Eigenschaft des Algorithmus von Dijkstra ausnutzen. Wenn man dies mit einer Durchschnittsanalyse verbindet, erreicht man sogar lineare erwartete Rechenzeit! Abschnitt 10.6 behandelt den Fall ganz beliebiger Kantenkosten, und in Abschnitt 10.7 befassen wir uns mit dem Problem, kürzeste Wege von jedem Knoten zu jedem anderen zu berechnen (*all pairs*). Wir zeigen, dass sich dieses Problem darauf reduzieren lässt, eine Instanz mit einem Startknoten und beliebigen Kantenkosten zu lösen und anschließend n Instanzen mit einem Startknoten und nichtnegativen Kantenkosten zu lösen. Im Zuge dieser Reduktion lernt der Leser auch das allgemein nützliche Konzept von Knotenpotenzialen kennen. Das Kapitel schließt mit Abschnitt 10.8, in dem das Problem von Kürzeste-Wege-*Anfragen* diskutiert wird.

10.1 Von Grundbegriffen zu einer allgemeinen Methode

Die Kostenfunktion für Kanten wird auf naheliegende Weise auf Wege erweitert. Die Kosten eines Weges sind die Summe der Kosten der Kanten, aus denen er besteht, d. h. die Kosten eines Wegs $p = \langle e_1, e_2, \ldots, e_k \rangle$ sind durch $c(p) = \sum_{1 \le i \le k} c(e_i)$ gegeben. Der leere Weg hat Kosten 0.

Wenn s und v Knoten sind, interessieren wir uns für einen *kürzesten Weg* von s nach v, also einen Weg von s nach v mit minimalen Kosten. Wir vermeiden hier absichtlich den bestimmten Artikel „den“, weil es mehrere verschiedene Wege geben kann, die in diesem Sinn kürzestmöglich sind. Gibt es überhaupt immer einen kürzesten Weg? Man beobachte, dass es unendlich viele Wege von s nach v geben kann. Wenn zum Beispiel der Weg $r = pCq$ von s nach v einen Kreis C enthält, können wir den Kreis beliebig oft durchlaufen und erhalten jedes Mal einen anderen Weg von s nach v, s. Abb. 10.1. Wir beschreiben dies genauer: p ist ein Weg von s nach u, Kreis C läuft von u nach u, und q ist ein Weg von u nach v. Betrachte den Weg $r^{(i)} = pC^iq$, der zunächst p durchläuft, dann C genau i-mal durchläuft, und schließlich q durchläuft. Die Kosten von $r^{(i)}$ sind $c(p) + i \cdot c(C) + c(q)$. Wenn C ein *negativer Kreis* ist, d. h., wenn $c(C) < 0$ gilt, dann gibt es keinen kürzesten Weg von s nach v. (Das liegt daran, dass die Menge $\{r^{(i)} \mid i \ge 0\} = \{c(p) + c(q) - i \cdot |c(C)| \mid i \ge 0\}$ negative Zahlen mit beliebig großem Betrag enthält.) Wir zeigen als Nächstes, dass es immer kürzeste Wege gibt, wenn der Graph G keine negativen Kreise enthält.

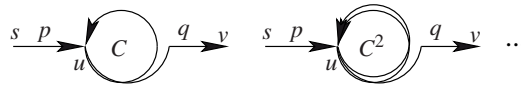


Abb. 10.1. Ein nicht-einfacher Weg pCq von s nach v.

Lemma 10.1. *Wenn G keine negativen Kreise enthält und v von s aus erreichbar ist, dann gibt es einen kürzesten Weg p_0 von s nach v, ja sogar einen kürzesten Weg p_0, der zudem einfach ist.*

Beweis. Sei p_0 ein *einfacher* Weg von s nach v mit minimalen Kosten unter den (endlich vielen) einfachen Wegen von s nach v. Angenommen, p_0 ist kein kürzester Weg von s nach v. Dann gibt es einen kürzeren, nicht-einfachen Weg r von s nach v. Weil r nicht einfach ist, können wir wie in Abb. 10.1 den Weg r als pCq schreiben, mit einem Kreis C und einem einfachen Weg pq. Dann gilt $c(p_0) \le c(pq)$, und daher $c(pq) + c(C) = c(r) < c(p_0) \le c(pq)$. Daraus folgt $c(C) < 0$, und wir haben einen negativen Kreis gefunden, im Widerspruch zur Voraussetzung. Also ist die Annahme falsch, und p_0 ist der gewünschte kürzeste Weg. □

Aufgabe 10.1. Verstärken Sie das eben bewiesene Lemma, indem Sie Folgendes zeigen: Für jeden von s aus erreichbaren Knoten v gibt es einen kürzesten Weg von s nach v genau dann wenn es keinen negativen Kreis gibt, der von s aus erreichbar ist und von dem aus v erreichbar ist.

Für zwei Knoten s und v definieren wir die „Kürzeste-Wege-Distanz“ oder kurz „Distanz“ $\mu(s,v)$ von s nach v als

$$\mu(s,v) := \begin{cases} +\infty & \text{wenn es keinen Weg von } s \text{ nach } v \text{ gibt,} \\ -\infty & \text{wenn } s \text{ von } t \text{ aus erreichbar ist,} \\ & \quad \text{aber kein kürzester Weg von } s \text{ nach } t \text{ existiert,} \\ c(p) & \text{für einen kürzesten Weg } p \text{ von } s \text{ nach } v \text{ sonst.} \end{cases}$$

Weil wir den Startknoten meistens mit s bezeichnen, benutzen wir auch die Abkürzung $\mu(v) := \mu(s,v)$. Man beachte Folgendes: Wenn v von s aus erreichbar ist, aber es keinen kürzesten Weg gibt, dann existieren Wege von s nach v mit beliebig stark negativen Kosten. Insofern ist es sinnvoll, in diesem Fall $\mu(v) = -\infty$ zu setzen. Kürzeste Wege haben einige hübsche Eigenschaften, die wir als Übungsaufgaben formulieren.

Aufgabe 10.2 (Teilwege von kürzesten Wegen). Zeigen Sie, dass Teilwege von kürzesten Wegen selbst kürzeste Wege sind, d. h., wenn ein aus drei Teilwegen p, q und r zusammengesetzter Weg pqr ein kürzester Weg ist, dann ist q auch ein kürzester Weg.

Aufgabe 10.3 (Kürzeste-Wege-Bäume). Nehmen wir an, dass alle n Knoten von s aus erreichbar sind und dass es keine negativen Kreise gibt. Zeigen Sie, dass es dann einen Baum T mit n Knoten gibt, in dem der Startknoten s die Wurzel bildet und alle Wege im Baum kürzeste Wege sind. *Hinweis*: Nehmen Sie zunächst an, dass es von s nach v nur einen einzigen kürzesten Weg gibt, und betrachten Sie den Teilgraphen T, der aus allen kürzesten Wegen besteht, die von s ausgehen. Benutzen Sie die vorige Übungsaufgabe, um zu zeigen, dass T ein Baum ist. Erweitern Sie das Ergebnis dann auf den Fall, in dem kürzeste Wege nicht unbedingt eindeutig sind.

Unsere Strategie, mit der kürzeste Wege von einem Startknoten s aus gefunden werden sollen, ist eine Verallgemeinerung der Breitensuche (BFS), die in Abb. 9.3 dargestellt ist. Wir arbeiten mit zwei Arrays vom Typ *NodeArray*, die d und *parent* heißen. Dabei enthält der Eintrag $d[v]$ unser gegenwärtiges Wissen über die Distanz von s nach v, und $parent[v]$ speichert den Vorgänger von v auf einem Weg von s nach v, der nach aktuellem Stand kürzestmöglich ist. Der Wert $d[v]$ wird meist als *Schätzdistanz* für v bezeichnet. Anfangs ist $d[s] = 0$ und $parent[s] = s$. Alle anderen Knoten haben Schätzdistanz „unendlich“ und keinen Vorgänger.

Der natürliche Ansatz zur Verbesserung von Abstandswerten ist, Abstandsinformation entlang von Kanten weiterzureichen. Wenn es einen Weg von s nach u mit Kosten höchstens $d[u]$ gibt und $e = (u,v)$ eine Kante ist, die von u ausgeht, dann gibt es einen Weg von s nach v mit Kosten nicht größer als $d[u] + c(e)$. Wenn diese Kosten kleiner sind als der beste bis dahin bekannte Distanzwert $d[v]$, aktualisieren wir $d[v]$ und $parent[v]$ entsprechend. Dieser Vorgang heißt *Kantenrelaxierung*:

Procedure *relax*$(e = (u,v) : Edge)$
  if $d[u] + c(e) < d[v]$ **then** $d[v] := d[u] + c(e)$; $parent[v] := u$

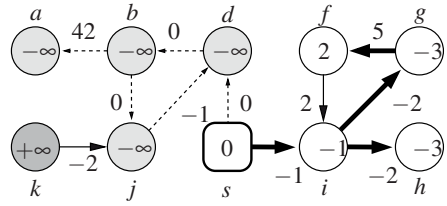


Abb. 10.2. Ein Graph mit Startknoten s und Kürzeste-Wege-Distanzen $\mu(v)$. Kantenkosten sind als Kantenmarkierungen angegeben, und die Distanzen stehen in den Knoten. Fett gezeichnete Kanten gehören zu kürzesten Wegen vom Startknoten s aus. Gestrichelte Kanten und hellgraue Knoten gehören zu einer unendlichen Menge von Wegen mit Startknoten s mit beliebig stark negativen Kosten. Der dunkle Knoten k ist von s aus unerreichbar.

(Hier wird mit dem Wert ∞ in naheliegender Weise gerechnet: es gilt $a < \infty$ und $\infty + a = \infty$ für alle Zahlen a, sowie $\infty \not< \infty$.)

Lemma 10.2. *Wenn nach einer beliebigen Folge von Kantenrelaxierungen $d[v] < \infty$ gilt, dann gibt es einen Weg von s nach v der Länge $d[v]$.*

Beweis. Wir benutzen Induktion über die Anzahl der Relaxierungen. Die Behauptung stimmt sicher vor der ersten Relaxierung: der leere Weg führt von s nach s und hat Länge $0 = d[s]$; alle anderen Knoten haben unendliche Schätzdistanz. Nun betrachten wir die Relaxierung einer Kante $e = (u,v)$. Nach Induktionsvoraussetzung gibt es einen Weg p der Länge $d[u]$ von s nach u und einen Weg der Länge $d[v]$ von s nach v. Wenn $d[u] + c(e) \geq d[v]$ gilt, ist nichts zu zeigen. Andernfalls ist pe ein Weg von s nach v der Länge $d[u] + c(e)$. □

Die Algorithmen in diesem Kapitel folgen alle der Strategie, wiederholt Kanten zu relaxieren, bis entweder für jeden Knoten v ein kürzester Weg von s nach v gefunden worden ist oder bis ein negativer Kreis entdeckt worden ist. Zum Beispiel geben die (Umkehrungen der) fett gezeichneten Kanten in Abb. 10.2 die *parent*-Information wieder, die nach einer genügend großen Anzahl von Kantenrelaxierungen gesammelt worden ist: die Knoten f, g, i und h sind von s aus über diese Kanten erreichbar und haben die korrekten $\mu(\cdot)$-Werte 2, -3, -1 und -3 erreicht. Die Knoten b, j und d bilden einen von s aus erreichbaren negativen Kreis, so dass ihre Kürzesten-Wege-Kosten $-\infty$ sind. Knoten a ist von diesem Kreis aus erreichbar und hat daher ebenfalls Abstand $\mu(a) = -\infty$. Knoten k ist von s aus unerreichbar; sein Abstand ist $+\infty$.

Was aber ist eine gute Folge von Kantenrelaxierungen? Sei $p = \langle e_1, \ldots, e_k \rangle$ ein Weg von s nach v. Wenn wir die Kanten dieses Wegs in der Reihenfolge von e_1 bis e_k relaxieren, gilt nachher $d[v] \leq c(p)$. Wenn p ein kürzester Weg von s nach v ist, kann (nach dem vorherigen Lemma) $d[v]$ nie kleiner als $c(p)$ werden, und daher gilt nach dieser Folge von Relaxierungen, dass $d[v] = c(p)$ ist.

Lemma 10.3 (Korrektheitskriterium). *Nach Ausführung einer Folge R von Relaxierungen, die einen kürzesten Weg $p = \langle e_1, e_2, \ldots, e_k \rangle$ von s nach v als Teilfolge ent-*

hält, (d. h., wenn es Indizes $t_1 < t_2 < \cdots < t_k$ mit $R[t_1] = e_1, R[t_2] = e_2, \ldots, R[t_k] = e_k$ gibt), gilt $d[v] = \mu(v)$. Weiter gilt: Durch die parent-Information an den Knoten wird ein Weg von s nach v der Länge μ definiert.

Beweis. Wir stellen im Folgenden R und p schematisch dar. Die erste Zeile gibt die Zeit an. Zum Zeitpunkt t_1 erfolgt die Relaxierung von e_1, zum Zeitpunkt t_2 die von e_2 usw.:

$$\begin{array}{lcl} & 1,2,\ldots,t_1,\ \ldots,\ t_2,\ \ldots\ldots,t_k,\ \ldots \\ R:=\langle & \ldots\ \ ,e_1,\ldots,e_2,\ldots\ldots,e_k,\ \ldots\rangle \\ p:= & \langle e_1, \quad e_2, \quad \ldots \quad ,e_k\rangle \end{array}$$

Weil p ein kürzester Weg ist, gilt $\mu(v) = \sum_{1\le j\le k} c(e_j)$. Für $i \in 1..k$ sei v_i der Zielknoten von e_i; weiter definieren wir $t_0 = 0$ und $v_0 = s$. Dann gilt nach Zeitpunkt t_i die Ungleichung

$$d[v_i] \le \sum_{1\le j\le i} c(e_j).$$

Dies zeigt man durch Induktion, wie folgt: Nach Zeitpunkt t_0, also ganz am Anfang, gilt die Ungleichung, weil $d[s] = 0$ gesetzt wird. Nun betrachte $i > 0$. Nach Induktionsvoraussetzung gilt nach der Relaxierung zum Zeitpunkt t_{i-1} die Ungleichung $d[v_{i-1}] \le \sum_{1\le j\le i-1} c(e_j)$. Weitere Relaxierungen zu Zeitpunkten zwischen t_{i-1} und t_i ändern daran nichts, da $d[v_{i-1}]$ dadurch nur kleiner werden kann. Durch die Relaxierung zum Zeitpunkt t_i wird sichergestellt, dass $d[v_i] \le d[v_{i-1}] + c(v_{i-1}, v_i)$ gilt. Durch Addieren der beiden letzten Ungleichungen ergibt sich die Induktionsbehauptung für Zeitpunkt t_i, was den Induktionsbeweis beendet. Aus der Ungleichung folgt, dass nach Zeitpunkt t_k die Beziehung $d[v] \le \mu(v)$ gilt. Nun kann aber nach Lemma 10.2 $d[v]$ nicht unter $\mu(v)$ fallen. Daher gilt nach Zeitpunkt t_k, und dann auch nach Ausführung der restlichen Relaxierungen in der Folge R, die Gleichung $d[v] = \mu(v)$.

Nun zeigen wir noch, dass die *parent*-Information kürzeste Wege beschreibt. Wir tun das unter der zusätzlichen Annahme, dass kürzeste Wege eindeutig sind, und überlassen den allgemeinen Fall dem Leser zur Übung. Aus den obigen Überlegungen folgt, dass nach den Relaxierungen in R die Gleichung $d[v_i] = \mu(v_i)$ für alle $i \in 1..k$ gilt. Zu irgendeinem Zeitpunkt wird also durch eine $relax(u, v_i)$-Operation in R der Wert $d[v_i]$ auf $\mu(v_i)$ gesetzt und $parent[v_i] = u$ gesetzt. (Man beachte, dass dies durchaus vor dem Zeitpunkt t_i geschehen sein könnte, aber nicht nachher.) Nach dem Beweis von Lemma 10.2 gibt es einen Weg der Länge $\mu(v_i)$ von s nach v_i, also einen kürzesten Weg, der mit der Kante (u, v_i) endet. Nach unserer Annahme sind kürzeste Wege eindeutig; also muss $u = v_{i-1}$ und damit $parent[v_i] = v_{i-1}$ gelten. Nachdem $d[v_i]$ auf den korrekten Wert $\mu(v_i)$ gesetzt worden ist, behält $parent[v_i]$ den Wert v_{i-1} bis zum Schluss. □

Aufgabe 10.4. Führen Sie den Beweis im zweiten Absatz des vorangehenden Beweises nochmals, aber ohne die Annahme, dass kürzeste Wege eindeutig bestimmt sind.

Aufgabe 10.5. Es sei S eine Auflistung der Kanten von G in einer beliebigen Anordnung; man bilde die Folge $S^{(n-1)}$ durch Nebeneinandersetzen von $n-1$ Kopien

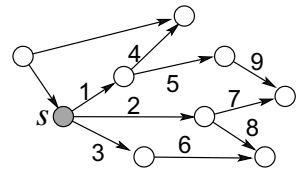


Abb. 10.3. Reihenfolge der Kantenrelaxierungen für die Berechnung der kürzesten Wege in einem DAG von einem Knoten s aus. Die topologische Sortierung der Knoten ist durch ihre x-Koordinate gegeben. Kanten, die von Knoten „links von“ s ausgehen, müssen überhaupt nicht relaxiert werden.

von S. Zeigen Sie: Nachdem die Relaxierungen $S^{(n-1)}$ durchgeführt worden sind, gilt $\mu(v) = d[v]$ für alle Knoten v mit $\mu(v) \neq -\infty$.

Wie wir in den folgenden Abschnitten sehen werden, lassen sich für azyklische Graphen und für Graphen ohne negative Kantengewichte effizientere Relaxierungsfolgen finden. Auf allgemeine Graphen kommen wir in Abschnitt 10.6 zurück.

10.2 Gerichtete azyklische Graphen

In einem gerichteten azyklischen Graphen (kurz „DAG“) gibt es keine gerichteten Kreise und daher auch keine negativen Kreise. Zudem haben wir in Abschnitt 9.2.1 festgestellt, dass die Knoten eines DAGs „topologisch angeordnet“ werden können, d. h. in eine Reihenfolge $\langle v_1, v_2, \ldots, v_n \rangle$ gebracht werden können, für die aus $(v_i, v_j) \in E$ stets $i < j$ folgt. Eine topologische Sortierung kann mit Tiefensuche in linearer Zeit $O(m+n)$ berechnet werden. Die Knoten entlang jedes Weges in einem DAG sind bezüglich der topologischen Sortierung aufsteigend geordnet. Nach Lemma 10.3 können wir also kürzeste-Wege-Distanzen berechnen, indem wir zuerst alle von v_1 ausgehenden Kanten relaxieren, dann die von v_2 ausgehenden Kanten, und so fort. (In Abb. 10.3 findet man ein Beispiel. Man kann sogar auf die Relaxierung der Kanten verzichten, die von Knoten ausgehen, die in der topologischen Sortierung vor s stehen.) Wenn man so vorgeht, wird jede Kante höchstens einmal relaxiert. Weil jede Relaxierung nur konstante Zeit kostet, ist die Ausführungszeit insgesamt $O(m+n)$.

Satz 10.4 *In azyklischen Graphen können kürzeste Wege von einem Knoten aus in Zeit* $O(m+n)$ *berechnet werden.*

Aufgabe 10.6 (Routenplanung für öffentlichen Personenverkehr). Das Problem, in einem öffentlichen Personenverkehrsnetz schnellste Verbindungen zu finden, kann als kürzeste-Wege-Problem in einem azyklischen Graphen modelliert werden. Man betrachte einen Bus oder einen Zug, der Station p zur Zeit t verlässt und seinen nächsten Halt p' zum Zeitpunkt t' erreicht. Diese Verbindung wird als Kante aufgefasst, die von Knoten (p,t) zu Knoten (p',t') verläuft. Weiter führen wir für jede Station p und unmittelbar aufeinanderfolgende (Ankunfts- oder Abfahrts-)Ereignisse

an p, etwa zu Zeiten t und t' mit $t < t'$, eine *Wartekante* von (p,t) nach (p,t') ein. (a) Zeigen Sie, dass der so definierte Graph ein DAG ist. (b) Ein zusätzlicher Knoten wird benötigt, der den Startpunkt des Reisenden als Orts- und Zeitangabe festlegt. Es muss auch eine Kante geben, die diesen Knoten mit dem Transportnetzwerk verbindet. Wie sollte diese Kante definiert sein? (c) Nehmen Sie an, ein Baum aus kürzesten Wegen vom Startknoten aus zu allen erreichbaren Knoten in diesem Personenverkehrsgraphen wurde berechnet. Wie findet man jetzt die Route, an der man eigentlich interessiert ist?

10.3 Nichtnegative Kantenkosten (Der Algorithmus von Dijkstra)

In diesem Abschnitt nehmen wir an, dass die Kantenkosten nicht negativ sind. Daraus folgt sofort, dass es keine negativen Kreise gibt und dass es kürzeste Wege für alle Knoten gibt, die von s aus erreichbar sind. Wir werden zeigen, dass bei einer klugen Wahl der Relaxierungsreihenfolge jede Kante nur einmal relaxiert werden muss.

Aber was ist die richtige Reihenfolge? Entlang eines kürzesten Weges steigen die Kürzeste-Wege-Distanzen an (genauer gesagt, sie fallen nicht). Das führt zu der Idee, dass die Knoten in der Reihenfolge wachsender Distanzen bearbeitet werden sollten. (Einen Knoten *bearbeiten* soll heißen, dass alle Kanten, die aus diesem Knoten herausführen, relaxiert werden.) Lemma 10.3 sagt uns, dass mit dieser Relaxierungsreihenfolge sichergestellt ist, dass die kürzesten Wege berechnet werden, zumindest wenn alle Kanten positive Länge haben. Im Algorithmus kennen wir natürlich die Kürzeste-Wege-Distanzen (noch) nicht, nur die *Schätzdistanzen* $d[v]$. Zum Glück stimmen für einen noch nicht bearbeiteten Knoten mit minimaler Schätzdistanz die wahre Distanz und die Schätzdistanz überein. Dies werden wir in Satz 10.5 feststellen. Es ergibt sich der in Abb. 10.4 dargestellte Algorithmus, der als „(Kürzeste-Wege-)Algorithmus von Dijkstra“ bekannt ist. In Abb. 10.5 ist der Ablauf an einem Beispiel dargestellt.

Im Grund ist der Algorithmus von Dijkstra nichts anderes ist als der Fäden-und-Knoten-Algorithmus aus der Einleitung zu diesem Kapitel, zumindest für ungerichtete Graphen. Wenn wir alle Fäden und Knoten auf einen Tisch legen und dann den

Algorithmus von Dijkstra
Alle Knoten sind „unbearbeitet“; initialisiere d und *parent*
while es gibt unbearbeitete Knoten mit Schätzdistanz $< +\infty$ **do**

$u :=$ ein unbearbeiteter Knoten mit minimaler Schätzdistanz
relaxiere alle von u ausgehenden Kanten (u,v);
damit ist u „bearbeitet“

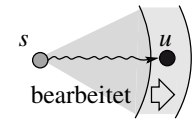


Abb. 10.4. Der Algorithmus von Dijkstra: Kürzeste Wege von einem Startknoten aus in Graphen mit nichtnegativen Kantengewichten

Operation	Schlange
insert(s)	$\langle(s,0)\rangle$
deleteMin$\rightsquigarrow(s,0)$	$\langle\rangle$
relax $s \xrightarrow{2} a$	$\langle(a,2)\rangle$
relax $s \xrightarrow{10} d$	$\langle(a,2),(d,10)\rangle$
deleteMin$\rightsquigarrow(a,2)$	$\langle(d,10)\rangle$
relax $a \xrightarrow{3} b$	$\langle(b,5),(d,10)\rangle$
deleteMin$\rightsquigarrow(b,5)$	$\langle(d,10)\rangle$
relax $b \xrightarrow{2} c$	$\langle(c,7),(d,10)\rangle$
relax $b \xrightarrow{1} e$	$\langle(e,6),(c,7),(d,10)\rangle$
deleteMin$\rightsquigarrow(e,6)$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{9} b$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{8} c$	$\langle(c,7),(d,10)\rangle$
relax $e \xrightarrow{0} d$	$\langle(d,6),(c,7)\rangle$
deleteMin$\rightsquigarrow(d,6)$	$\langle(c,7)\rangle$
relax $d \xrightarrow{4} s$	$\langle(c,7)\rangle$
relax $d \xrightarrow{5} b$	$\langle(c,7)\rangle$
deleteMin$\rightsquigarrow(c,7)$	$\langle\rangle$

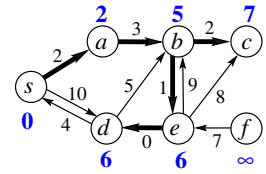


Abb. 10.5. Ablauf des Algorithmus von Dijkstra auf dem *rechts* angegebenen Graphen. Die fett gezeichneten Kanten bilden den Kürzeste-Wege-Baum; die fett geschriebenen Zahlen geben die Distanzen an. Die Tabelle *links* stellt die ausgeführten Schritte dar. Die (Warte-)*Schlange* enthält stets die Paare $(v,d[v])$ für Knoten v, die gefunden, aber noch nicht bearbeitet worden sind. Dabei heißt ein Knoten *gefunden*, wenn seine Schätzdistanz kleiner als $+\infty$ ist. Anfangs ist nur s gefunden und nicht bearbeitet. Die Aktionen des Algorithmus stehen in der ersten Spalte; der Inhalt der Schlange nach der jeweiligen Aktion in der zweiten.

Startknoten allmählich immer höher heben, lösen sich die anderen Knoten von der Tischplatte in der Reihenfolge ihrer Kürzeste-Wege-Distanzen.

Satz 10.5 *Der Algorithmus von Dijkstra löst das Kürzeste-Wege-Problem mit einem Startknoten für Graphen mit nichtnegativen Kantenkosten.*

Beweis. Der Beweis besteht aus zwei Schritten. Im ersten Schritt zeigen wir, dass alle von s aus erreichbaren Knoten irgendwann bearbeitet werden. Im zweiten Schritt zeigen wir, dass in dem Moment, in dem ein Knoten bearbeitet wird, seine Schätzdistanz und seine Kürzeste-Wege-Distanz übereinstimmen.

Wir sagen, ein Knoten wird *gefunden*, wenn seine Schätzdistanz einen Wert $< +\infty$ erhält. Der Algorithmus stellt offenbar sicher, dass jeder Knoten v, der gefunden wird, irgendwann auch bearbeitet wird.

Erster Schritt: Nach der eben gemachten Bemerkung genügt es zu zeigen, dass alle von s aus erreichbaren Knoten irgendwann gefunden werden. Dazu betrachten wir einen Weg $p = \langle s = v_1, v_2, \ldots, v_k = v\rangle$ von s zu einem Knoten v, und zeigen durch Induktion über i, dass v_i gefunden wird. Knoten $s = v_1$ wird gefunden, weil er durch die Initialisierung Schätzdistanz $d[s] = 0 < +\infty$ erhält. Nun sei $i > 1$. Nach Induktionsvoraussetzung wird v_{i-1} gefunden, also $d[v_{i-1}] < +\infty$ gesetzt. Irgendwann später wird v_{i-1} bearbeitet, und das heißt, dass $\mathit{relax}(v_{i-1}, v_i)$ ausgeführt wird. Spätestens nach dieser Operation gilt $d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) < +\infty$, also wird v_i gefunden.

Zweiter Schritt: Durch Induktion über die Schleifendurchläufe („Runden“) beweisen wir folgende Behauptung: Wenn in Runde t Knoten u bearbeitet wird, dann

gilt $d[u] \leq \mu[u]$. (Man beachte, dass nach Lemma 10.2 die Ungleichung $d[u] \geq \mu[u]$ immer gilt.) Die Behauptung gilt für $t = 1$, weil in Runde 1 Knoten s bearbeitet wird, mit $d[s] = 0 = \mu(s)$. Nun betrachten wir eine Runde $t > 1$ und den in dieser Runde bearbeiteten Knoten $u \neq s$. Wir wählen einen beliebigen Weg $p = \langle s = v_1, v_2, \ldots, v_k = u \rangle$ von s nach u. Sei v_i mit $1 < i \leq k$ der erste Knoten auf diesem Weg, der *nicht* vor Runde t bearbeitet wird. Dann wird Knoten v_{i-1} vor Runde t bearbeitet. Dabei wird die Operation $\mathit{relax}(v_{i-1}, v_i)$ ausgeführt. Nachher gilt:

$$d[v_i] \leq d[v_{i-1}] + c(v_{i-1}, v_i) \leq \mu(v_{i-1}) + c(v_{i-1}, v_i).$$

(Die erste Ungleichung wird durch die Operation $\mathit{relax}(v_{i-1}, v_i)$ erzwungen; die zweite gilt wegen der Induktionsvoraussetzung.) Der Weg $\langle v_1, \ldots, v_{i-1} \rangle$ von s nach v_{i-1} hat Kosten mindestens $\mu(v_{i-1})$, also gilt

$$\mu(v_{i-1}) + c(v_{i-1}, v_i) \leq c(\langle v_1, \ldots, v_{i-1} \rangle) + c(v_{i-1}, v_i) = c(\langle v_1, \ldots, v_i \rangle).$$

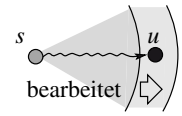
Weil $\langle v_1, v_2, \ldots, v_i \rangle$ ein Anfangsstück von p ist und Kantenkosten nicht negativ sind, gilt weiter $c(\langle v_1, \ldots, v_i \rangle) \leq c(p)$. Durch Kombinieren der letzten drei Ungleichungen ergibt sich, dass nach der Operation $\mathit{relax}(v_{i-1}, v_i)$ die Ungleichung $d[v_i] \leq c(p)$ gilt. Auch wenn eventuell $d[v_i]$ noch sinkt, gilt dies auch zu Beginn von Runde t. Weil der Algorithmus in Runde t Knoten u als den Knoten mit minimaler Schätzdistanz zur Bearbeitung auswählt, muss $d[u] \leq d[v_i]$, also $d[u] \leq c(p)$ gelten. Da p beliebig war, folgt $d[u] \leq \mu(u)$, also die Induktionsbehauptung. □

Aufgabe 10.7. Es seien $v_1, v_2, \ldots$ die Knoten in der Reihenfolge, in der sie bearbeitet werden. Zeigen Sie: $\mu(v_1) \leq \mu(v_2) \leq \ldots$, d. h., die Knoten werden in einer Reihenfolge von (schwach) steigenden Kürzeste-Wege-Distanzen bearbeitet.

Aufgabe 10.8 (Prüfen von Kürzeste-Wege-Distanzen). Wir nehmen an, dass alle Kantenkosten positiv sind und alle Knoten von s aus erreichbar sind. Weiter soll d ein Knotenarray mit nichtnegativen reellen Einträgen sein, für die $d[s] = 0$ und $d[v] = \min_{(u,v) \in E} d[u] + c(u, v)$ gilt, für alle $v \neq s$. Zeigen Sie, dass dann für alle v die Gleichheit $d[v] = \mu(v)$ gilt. Stimmt diese Behauptung auch noch, wenn es Kanten mit Kosten 0 gibt?

Wir wenden uns nun der Implementierung des Algorithmus von Dijkstra zu. Alle unbearbeiteten Knoten werden in einer adressierbaren Prioritätswarteschlange gespeichert (s. Abschnitt 6.2), wobei die Schätzdistanzen als Schlüssel dienen. Die Operation *deleteMin* liefert dann immer den nächsten zu bearbeitenden Knoten. Zur Adressierung von Einträgen bei der *decreaseKey*-Operation benutzen wir hier Knotennummern anstelle von Griffen. Ausgehend von einer gewöhnlichen Prioritätswarteschlange braucht man für die Implementierung dieser (*NodePQ* genannten) Datenstruktur nur ein zusätzliches Knotenarray, mit dessen Hilfe Knotennummern in Griffe übersetzt werden. Wenn die Prioritätswarteschlange Baumstruktur hat, wie etwa bei Fibonacci-Heaps oder Pairing-Heaps, kann man auch die Einträge der *NodePQ*, d. h. die Baumknoten, direkt in einem Knotenarray speichern. Ein Aufruf $\mathit{decreaseKey}(v)$ aktualisiert die *NodePQ* unter der Annahme, dass unmittelbar vorher der Schlüssel $d[v]$ verringert wurde.

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray          // gibt (d, parent) zurück
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ ∪ {∞}                     // Schätzdistanzen
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    parent[s] := s                                           // Schleife signalisiert Wurzel
    Q : NodePQ                                               // unbearbeitete gefundene Knoten
    d[s] := 0;   Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin                                     // es gilt d[u] = μ(u)
        foreach edge e = (u,v) ∈ E do
            if d[u] + c(e) < d[v] then                       // relaxiere
                d[v] := d[u] + c(e)
                parent[v] := u                               // aktualisiere Baum
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d, parent)
```



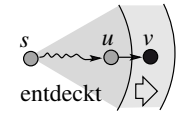


Abb. 10.6. Pseudocode für den Algorithmus von Dijkstra

Wir erhalten den in Abb. 10.6 dargestellten Algorithmus. Als Nächstes analysieren wir seine Rechenzeit in Abhängigkeit von den Zeiten, die für die Operationen auf der Prioritätswarteschlange anfallen. Die Initialisierung der Arrays d und *parent* und die Erstellung der Prioritätswarteschlange $Q = \{s\}$ kosten Zeit $O(n)$. Die Überprüfung der Bedingung $Q = \emptyset$ und die Schleifensteuerung benötigt in jedem Durchlauf durch die while-Schleife konstante Zeit, d. h. insgesamt Zeit $O(n)$. Jeder von s aus erreichbare Knoten wird genau einmal aus der Prioritätswarteschlange entnommen. Jeder erreichbare Knoten, der von s verschieden ist, wird auch genau einmal eingefügt. Daher gibt es höchstens n *deleteMin*- und *insert*-Operationen. Weil jeder Knoten höchstens einmal bearbeitet wird, wird jede Kante höchstens einmal relaxiert, und daher kann es höchstens m *decreaseKey*-Operationen geben. Als Gesamtausführungszeit für den Algorithmus von Dijkstra ergibt sich

$$T_{Dijkstra} = O\big(m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n))\big)\,,$$

wobei $T_{deleteMin}$, T_{insert} und $T_{decreaseKey}$ die Ausführungszeiten für *deleteMin*, *insert* bzw. *decreaseKey* angeben. Man beachte, dass diese Ausführungszeiten von der Größe $|Q| = O(n)$ der Prioritätswarteschlange abhängen.

Aufgabe 10.9. Gegeben seien n, m mit $n \le m \le n(n-1)/2$. Konstruieren Sie einen Graphen G mit n Knoten und m Kanten und einen Satz nichtnegativer Kantengewichte, so dass bei der Ausführung des Algorithmus von Dijkstra $m-(n-1)$ der Kantenrelaxierungen eine *decreaseKey*-Operation auslösen!

In seiner 1959 erschienenen Originalarbeit schlug Dijkstra die folgende Implementierung für die Prioritätswarteschlange vor: Zähle die Anzahl der gefundenen,

aber noch nicht bearbeiteten Knoten mit, und verwalte zwei Arrays, jeweils mit der Knotenmenge als Indexmenge – ein Array d, das die Schätzdistanzen speichert, und ein Array, das für jeden Knoten angibt, ob er noch nicht gefunden, gefunden und unbearbeitet, oder bearbeitet ist. Dann kosten *insert* und *decreaseKey* Zeit $O(1)$. Eine *deleteMin*-Operation kostet Zeit $O(n)$, weil dafür das ganze Array durchlaufen werden muss, um einen gefundenen, aber unbearbeiteten Knoten mit minimaler Schätzdistanz zu finden. Die Gesamtzeit wird damit

$$T_{Dijkstra59} = O\left(m + n^2\right) .$$

In der Zeit nach dem Erscheinen von Dijkstras Originalarbeit wurden viel bessere Prioritätswarteschlangen gefunden. Wenn wir einen Binärheap bzw. einen Fibonacci-Heap verwenden (beide in Abschnitt 6.2 beschrieben), erhalten wir

$$T_{DijkstraBHeap} = O((m+n)\log n)$$

bzw.

$$T_{DijkstraFibonacci} = O(m + n\log n) .$$

Asymptotisch ist die Implementierung mit einem Fibonacci-Heap überlegen, außer für sehr dünn besetzte Graphen mit $m = O(n)$. In der Praxis erhält man mit Fibonacci-Heaps nicht die schnellste Implementierung, weil die Rechenzeiten größere konstante Faktoren enthalten und weil es normalerweise viel weniger *decreaseKey*-Operationen gibt als von den worst-case-Abschätzungen vorhergesagt. Diese experimentelle Beobachtung wird im nächsten Abschnitt durch eine theoretische Analyse untermauert.

10.4 *Durchschnittsanalyse des Algorithmus von Dijkstra

In diesem Abschnitt zeigen wir, dass der Algorithmus von Dijkstra im erwarteten Fall $O(n\log(m/n))$ *decreaseKey*-Operationen ausführt.

Unser Zufallsmodell sieht folgendermaßen aus: Der Graph G und der Startknoten s sind beliebig. Für jeden Knoten v liegt eine (Multi-)Menge $C(v)$ von $indegree(v)$ vielen nichtnegativen reellen Zahlen vor. Bis hierher ist alles beliebig. Der Zufall kommt jetzt: Wir nehmen an, dass für jeden Knoten v die Zahlen in $C(v)$ zufällig den *Eingangskanten* von v als Kosten zugewiesen werden. Der Wahrscheinlichkeitsraum besteht also aus den $\prod_{v\in V} indegree(v)!$ möglichen Kombinationen von solchen Zuordnungen. Jede von ihnen hat dieselbe Wahrscheinlichkeit. Es sei betont, dass dieses Modell sehr allgemein ist. Insbesondere enthält es als Spezialfall die Situation, dass die Kosten jeder Kante unabhängig aus einer festen Verteilung gezogen werden.

Satz 10.6 *Unter den genannten Annahmen ist die erwartete Anzahl der vom Algorithmus von Dijkstra ausgeführten decreaseKey-Operationen* $O(n\log(m/n))$.

Beweis. Der hier vorgestellte Beweis stammt von Noshita [163].

Bevor wir mit dem Beweis beginnen, benötigen wir noch eine technische Bemerkung zur Funktionsweise der verwendeten Prioritätswarteschlange. Wenn es zu einem Zeitpunkt mehrere Knoten gibt, die alle dieselbe minimale Schätzdistanz haben, ist es für die Analyse der Rechenzeit im schlechtesten Fall unerheblich, welcher davon von der *deleteMin*-Operation zurückgegeben wird, und dies wird von der Spezifikation des Datentyps „Prioritätswarteschlange“ auch nicht festgelegt. Für den nun folgenden Beweis müssen wir verlangen, dass diese Entscheidung *konsistent* getroffen wird, zum Beispiel in folgender Weise: Wenn mehrere Knoten dieselbe aktuell minimale Schätzdistanz haben, hat der mit der kleinsten Nummer Vorrang. Man kann dies bei einer beliebigen Realisierung der Prioritätswarteschlange, etwa einem Binärheap, dadurch erreichen, dass man als Schlüssel für v nicht einfach die Schätzdistanz $d[v]$, sondern das Paar $(d[v], v)$ benutzt und als Ordnung auf diesen Schlüsseln die lexikographische Ordnung verwendet.

Wir betrachten zunächst einen beliebigen Knoten $v \neq s$ und nehmen an, dass für jeden Knoten $u \neq v$ den Eingangskanten von u ihre Kosten schon fest zugewiesen worden sind. Nur die Zuordnung der Kosten in $C(v)$ zu den Eingangskanten von v ist offen und wird zufällig bestimmt. Wir wollen zeigen, dass die erwartete Anzahl von *decreaseKey*(v)-Operationen durch $\ln(\mathit{indegree}(v))$ beschränkt ist. Dazu leiten wir eine Beziehung zwischen der Anzahl dieser *decreaseKey*(v)-Operationen und der Anzahl der Zwischenmaxima in einer Zufallsfolge der Länge *indegree*(v) her (siehe Abschnitt 2.7.2).

Die Hauptschwierigkeit im Beweis liegt darin, dass möglicherweise die Reihenfolge, in der die Eingangskanten von v relaxiert werden, von der Zuordnung der Kosten zu diesen Kanten abhängt. Die entscheidende Beobachtung ist, dass bis zu dem Zeitpunkt, zu dem v bearbeitet wird, diese Reihenfolge immer gleich ist. Dazu betrachten wir parallel den Ablauf des Algorithmus von Dijkstra auf $G - v$ (also G ohne v und die mit v inzidenten Kanten) und auf G mit einer beliebig gewählten Zuordnung der Kosten bei v. Bevor (im letzteren Ablauf) v bearbeitet wird, werden in beiden Abläufen genau dieselben Knoten in genau derselben Reihenfolge bearbeitet. Dies gilt deswegen, weil bis zur Bearbeitung von v keine Ausgangskante von v relaxiert wird, also Knoten v und seine Schätzdistanz $d[v]$ auf den Ablauf und die Schätzdistanzen anderer Knoten keinen Einfluss haben. Wegen der eingangs gemachten Bemerkung zur Konsistenz der Arbeitsweise der Prioritätswarteschlange ändert auch das Vorhandensein von v in dieser Datenstruktur nichts an den Ergebnissen der *deleteMin*-Aufrufe. Man beachte aber, dass der Zeitpunkt, zu dem v bearbeitet wird, durchaus von der Zuweisung der Kosten zu den Eingangskanten von v abhängen kann. Wenn v aber erst einmal bearbeitet worden ist, gibt es keine *decreaseKey*(v)-Operation mehr.

Seien $u_1, \ldots, u_k$, für $k = \mathit{indegree}(v)$, die Startknoten der Eingangskanten von v in der Reihenfolge, in der sie beim Ablauf des Algorithmus von Dijkstra auf $G - v$ bearbeitet werden, und sei $\mu'(u_i)$ der Abstand von s nach u_i in $G - v$. Knoten u_i, die in $G - v$ von s aus nicht erreichbar sind, haben unendlichen Abstand und stehen in dieser Liste ganz hinten. Nach Übungsaufgabe 10.7 gilt dann $\mu'(u_1) \leq \cdots \leq \mu'(u_k)$. Im Ablauf des Algorithmus auf G werden die Eingangskanten von v in der Reihenfolge $e_1 = (u_1, v), e_2 = (u_2, v), \ldots$ relaxiert, bis zu dem Zeitpunkt, an dem

v bearbeitet wird. Für die dabei auftretenden Distanzen $\mu(u_i)$ gilt $\mu(u_i) = \mu'(u_i)$. Wenn nun die Relaxierung von e_i eine *decreaseKey*(v)-Operation auslöst, dann gilt $i \geq 2$ (die Relaxierung von e_1 löst kein *decreaseKey*(v), sondern ein *insert*(v) aus), es gilt

$$\mu'(u_i) + c(e_i) < \min_{j<i}(\mu'(u_j) + c(e_j))$$

und v ist bisher noch nicht bearbeitet worden. Weil für $1 \leq j < i$ die Ungleichung $\mu'(u_j) \leq \mu'(u_i)$ gilt, folgt

$$c(e_i) < \min_{j<i} c(e_j).$$

Das bedeutet, dass die Relaxierung von e_i nur dann eine *decreaseKey*(v)-Operation auslösen kann, wenn $i \geq 2$ gilt und $c(e_i)$ in der Folge $c(e_1), \ldots, c(e_k)$ ein *Zwischenminimum* ist. (Zwischenminima sind analog zu Zwischenmaxima definiert.) Daher kann die Anzahl der *decreaseKey*(v)-Operationen nicht größer sein als die Anzahl der Zwischenminima in dieser Folge minus 1. In Abschnitt 2.7 haben wir gesehen, dass die erwartete Anzahl von Zwischenmaxima in einer zufällig angeordneten Folge der Länge k durch H_k beschränkt ist. Diese Schranke gilt auch für die Zwischenminima. Daher ist die erwartete Anzahl von *decreaseKey*(v)-Operationen nicht größer als $H_k - 1$, was wiederum durch $\ln k = \ln(\mathit{indegree}(v))$ beschränkt ist, siehe (A.12).

Wegen der Linearität des Erwartungswerts können wir diese Schranken über alle $v \in V$ aufaddieren und auf diese Weise für die erwartete Gesamtanzahl der *decreaseKey*-Operationen die Schranke

$$\sum_{v \in V} \ln(\mathit{indegree}(v)) \leq n \ln \frac{m}{n}$$

erhalten. Die letzte Ungleichung folgt dabei aus der Tatsache, dass die Logarithmusfunktion konkav ist (s. (A.15)). □

Wir schließen, dass die erwartete Rechenzeit $O(m + n \log(m/n) \log n)$ ist, wenn man die Implementierung der Prioritätswarteschlange mit Binärheaps benutzt. Für genügend dichte Graphen ($m > n \log n \log\log n$) erhalten wir eine erwartete Ausführungszeit, die linear in der Eingabegröße ist.

Aufgabe 10.10. Zeigen Sie: Aus $m = \Omega(n \log n \log\log n)$ folgt $E[T_{\mathit{DijkstraBHeap}}] = O(m)$.

10.5 Monotone ganzzahlige Prioritätswarteschlangen

Der Entwurf des Algorithmus von Dijkstra zielt darauf ab, dass die Knoten in der Reihenfolge nichtfallender Schätzdistanzen bearbeitet werden. Daher genügt eine monotone Prioritätswarteschlange (s. Kap. 6) für die Implementierung. Man weiß nicht, ob die Eigenschaft der Monotonie im Fall allgemeiner reeller Kantenkosten zur Beschleunigung ausgenutzt werden kann. Wenn aber die Eigenschaft hinzukommt, dass die Kantenkosten ganzzahlig sind, sind signifikante Einsparungen möglich. In

diesem Abschnitt nehmen wir daher an, dass die Kantenkosten ganze Zahlen in einem Intervall $0..C$ sind, für eine natürliche Zahl C. Die Zahl C soll bekannt sein, wenn die Prioritätswarteschlange initialisiert wird.

Weil wir uns auf kürzeste Wege ohne Knotenwiederholungen beschränken können, ist die Länge von kürzesten Wegen durch $C(n-1)$ beschränkt. Der Bereich von Werten, die gleichzeitig in der Schlange stehen können, ist noch kleiner. Sei *min* der letzte aus der Schlange entnommene Wert (Null vor Beginn des Algorithmus). Während des Ablaufs des Algorithmus von Dijkstra gilt stets die Invariante, dass alle Wert in der Schlange im Bereich $min..min+C$ liegen. Diese Invariante gilt sicher nach der ersten Einfügung. Durch eine Ausführung von *deleteMin* kann sich *min* erhöhen. Weil alle Werte in der Schlange durch C plus den alten Wert von *min* beschränkt sind, gilt dies erst recht für den neuen Wert von *min*. Kantenrelaxierungen fügen Schlüssel der Form $d[u]+c(e) = min+c(e) \in min..min+C$ ein.

10.5.1 Behälterschlangen

Eine Behälterschlange (engl.: *bucket queue*) ist ein zirkuläres Array B von $C+1$ vielen doppelt verketteten Listen (s. Abb. 10.7 und 3.8). Die natürlichen Zahlen werden „um das zirkuläre Array herumgewickelt“ in dem Sinn, dass alle Zahlen der Form $i+(C+1)j$ auf den Index i abgebildet werden. Ein Knoten v, der mit Schätzdistanz $d[v]$ in der Prioritätswarteschlange Q steht, wird in „Behälter“ $B[d[v] \bmod (C+1)]$ gespeichert. Weil die Schätzdistanzen in Q stets im Bereich $min..min+C$ liegen, haben alle Knoten, die im gleichen Behälter liegen, die *gleiche* Schätzdistanz.

Bei der Initialisierung werden $C+1$ leere Listen erzeugt. Eine Operation $insert(v)$ fügt v in $B[d[v] \bmod (C+1)]$ ein. Eine Operation $decreaseKey(v)$ entfernt Knoten v aus der Liste, in der er bisher stand, und fügt ihn in $B[d[v] \bmod (C+1)]$ ein. Daher kosten *insert* und *decreaseKey* konstante Zeit, wenn die Behälter als doppelt verkettete Listen implementiert werden.

Eine *deleteMin*-Operation untersucht zuerst den Behälter $B[min \bmod (C+1)]$. Falls dieser Behälter leer ist, wird *min* inkrementiert und es wird von vorn begonnen. Auf diese Weise haben alle *deleteMin*-Operationen zusammen Kosten $O(n+nC) = O(nC)$, weil *min* höchstens nC-mal inkrementiert wird und höchstens n Einträge aus der Prioritätswarteschlange entnommen werden. Wenn wir die Operationskosten für die Prioritätswarteschlange in unsere allgemeine Schranke für die Kosten des Algorithmus von Dijkstra einsetzen, erhalten wir

$$T_{DijkstraBucket} = O(m+nC).$$

***Aufgabe 10.11 (Behälterschlangen in der Variante von Dinitz [63]).** Nehmen Sie an, dass die Kantenkosten positive reelle Zahlen im Intervall $[c_{\min}, c_{\max}]$ sind. Erklären Sie, wie man in Zeit $O(m+nc_{\max}/c_{\min})$ kürzeste Wege von einem Startknoten aus finden kann. *Hinweis*: Benutzen Sie Behälter der „Weite“ $c_{\min}$, d. h., jeder Behälter enthält Knoten mit Schätzdistanzen aus einem bestimmten Intervall der Länge $c_{\min}$. Zeigen Sie, dass alle Knoten v im nichtleeren Behälter mit den kleinsten Schätzdistanzen die Gleichung $d[v] = \mu(v)$ erfüllen.

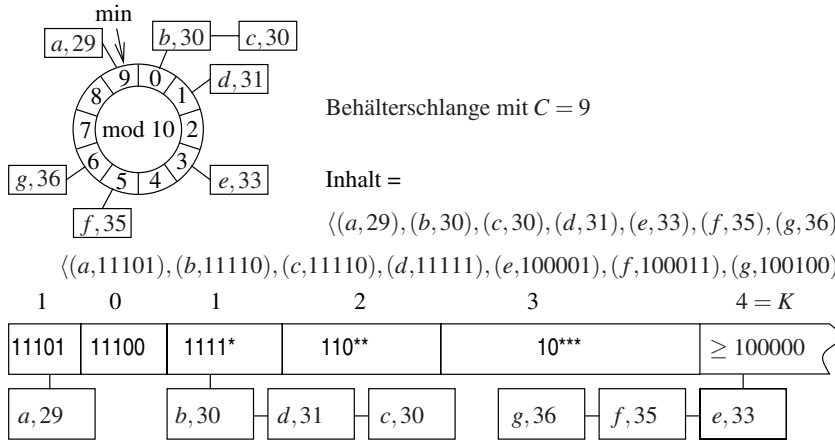


Abb. 10.7. Beispiel einer Behälterschlange (*oben*) und eines entsprechenden Radix-Heaps (*unten*). Wegen $C = 9$ haben wir $K = 1 + \lfloor \log C \rfloor = 4$. Die Bitmuster in den Behältern des Radix-Heaps geben an, welche Schlüssel ihnen zugeordnet werden

10.5.2 *Radix-Heaps

Radix-Heaps [9] verbessern die Implementierung von Prioritätswarteschlangen als Behälterschlangen, indem sie Behälter unterschiedlicher Weite benutzen. Für Schätzdistanzen in der Nähe von *min* benutzt man enge Behälter, für solche viel größer als *min* weite Behälter. In diesem Abschnitt zeigen wir, dass dieser Ansatz eine Version des Algorithmus von Dijkstra liefert, die Rechenzeit

$$T_{DijkstraRadix} = \mathrm{O}(m + n \log C)$$

aufweist.

Radix-Heaps arbeiten mit der Binärdarstellung der Schätzdistanzen. Wir erklären dazu, was wir unter dem *höchstwertigen unterscheidenden Index* (engl.: *most significant distinguishing index*) zweier Zahlen a und b verstehen wollen. Dies ist der größte Index, an dem sich die Binärdarstellungen unterscheiden, d. h., wenn $a = \sum_{i \geq 0} \alpha_i 2^i$ und $b = \sum_{i \geq 0} \beta_i 2^i$ die Binärdarstellungen angeben, ist der höchstwertige unterscheidende Index $msd(a,b)$ das größte i mit $\alpha_i \neq \beta_i$. Für $a = b$ setzen wir $msd(a,b) = -1$. Wenn $a < b$ gilt, hat a ein 0-Bit in Position $i = msd(a,b)$ und b ein 1-Bit.

Ein Radix-Heap besteht aus einem Array von Behältern $B[-1]$, $B[0]$, …, $B[K]$, mit $K = 1 + \lfloor \log C \rfloor$. Die Einträge der Prioritätswarteschlange werden gemäß der folgenden Regel auf die Behälter verteilt:

Eintrag v wird in Behälter $B[i]$ gespeichert, für $i = \min\{msd(min, d[v]), K\}$.

Diese Regel heißt die Behälterschlangen-Invariante. Ein Beispiel ist in Abb. 10.7 angegeben. Wir bemerken, dass Behälter $B[i]$ leer ist, wenn *min* in Position i ein 1-Bit hat, für $0 \leq i < K$. Dies gilt, weil jede Schätzdistanz $d[v]$ mit $i = msd(min, d[v])$

ein 0-Bit in Position i hätte und damit kleiner als *min* wäre, was der Definition von *min* widerspricht.

Wie kann man $i := msd(a,b)$ berechnen? Wir beobachten zunächst, dass für $a \neq b$ im bitweisen XOR $a \oplus b$ von a und b das höchstwertige 1-Bit an Stelle i steht. Dieses Binärwort stellt also eine natürliche Zahl zwischen 2^i (einschließlich) und 2^{i+1} (ausschließlich) dar. Daher gilt $msd(a,b) = \lfloor \log(a \oplus b) \rfloor$, weil $\log(a \oplus b)$ eine reelle Zahl mit ganzzahligem Anteil i ist, und die untere Gaußklammer den ganzzahligen Teil herauszieht. Viele Prozessoren ermöglichen die direkte Berechnung von *msd* durch Maschinenbefehle.[2] Alternativ können wir vorberechnete Tabellen oder andere Verfahren benutzen. Von hier an nehmen wir an, dass sich *msd* in konstanter Zeit berechnen lässt.

Wir wenden uns nun den Operationen der Prioritätswarteschlange zu. Die Initialisierung, *insert* und *decreaseKey* werden genau wie bei Behälterschlangen realisiert. Der einzige Unterschied ist, dass Behälterindizes gemäß der Behälterschlangen-Invariante berechnet werden.

Bei der Ausführung einer *deleteMin*-Operation wird als Erstes das kleinste i gesucht, für das Behälter $B[i]$ nicht leer ist. Wenn $i = -1$ ist, wird ein beliebiger Eintrag aus $B[-1]$ entnommen und zurückgegeben. Wenn $i \geq 0$ ist, wird Behälter $B[i]$ durchmustert und *min* wird auf die kleinste Schätzdistanz gesetzt, die im Behälter vorkommt. Weil sich *min* geändert hat, muss die Behälterschlangen-Invariante wiederhergestellt werden. Eine für die Effizienz von Radix-Heaps entscheidende Beobachtung besteht darin, dass nur Knoten in Behälter i betroffen sind. Wir sehen uns weiter unten genauer an, was genau für diese Knoten getan werden muss. Zunächst betrachten wir die Behälter $B[j]$ mit $j \neq i$. Die Behälter $B[j]$ mit $j < i$ sind leer. Wenn $i = K$ gilt, dann gibt es kein j mit $j > K$. Wenn $i < j \leq K$ gilt, erfüllt jeder Schlüssel a in $B[j]$ auch für den neuen Wert *min* die Gleichheit $msd(a, min) = j$, weil alter und neuer Wert von *min* in den Bitpositionen größer als i übereinstimmen.

Was passiert mit den Einträgen in $B[i]$? Sie werden in den passenden neuen Behälter verschoben. Daher kostet ein *deleteMin* konstante Zeit, wenn $i = -1$ gilt, und Zeit $O(i + |B[i]|) = O(K + |B[i]|)$, wenn $i \geq 0$ gilt. Lemma 10.7 unten zeigt, dass jeder Knoten in Behälter $B[i]$ in einen Behälter mit kleinerem Index verschoben wird. Diese Beobachtung macht es möglich, die Kosten der *deleteMin*-Operationen mit Hilfe einer amortisierten Analyse abzuschätzen. Als unsere Kosteneinheit (ein Jeton) benutzen wir die Zeit, die benötigt wird, einen Knoten von einem Behälter in einen anderen zu verschieben.

Wir verlangen $K + 1$ Jetons für die Operation *insert*(v), und heften diese $K + 1$ Jetons an Knoten v. Sie werden dafür benutzt, um für die Verlagerung von v in Behälter mit niedrigeren Indizes zu bezahlen, die sich bei einer *deleteMin*-Operation ergeben kann. Ein Knoten v startet in einem Behälter j mit $j \leq K$, um schließlich in Behälter -1 zu landen, und bewegt sich zwischenzeitlich nie zurück in Behälter mit höheren Nummern. Wir bemerken, dass auch *decreaseKey*(v)-Operationen nie Knoten in Behälter mit höheren Indizes verlagern. Daher können die $K + 1$ Je-

[2] $\oplus$ ist selbst ein Maschinenbefehl, und $\lfloor \log x \rfloor$ ist der Exponent in der Gleitkommadarstellung von x.

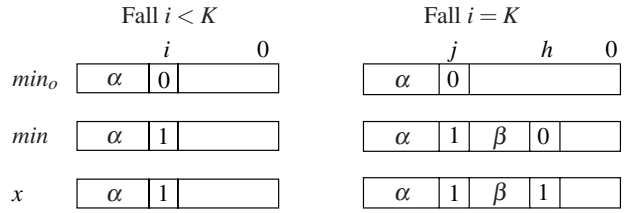


Abb. 10.8. Die Struktur der Schlüssel, die in Lemma 10.7 eine Rolle spielen. Im Beweis wird gezeigt, dass die ersten $j-K$ Ziffern von β Nullen sind.

tons für alle Knotenbewegungen bezahlen, die von *deleteMin*-Operationen ausgelöst werden. Für jede *deleteMin*-Operation fallen weiterhin Kosten von $O(K)$ an, die benötigt werden, einen nichtleeren Behälter zu finden. Mit amortisierten Kosten $K+1+O(1) = O(K)$ für ein *insert* und $O(1)$ für ein *decreaseKey* erhalten wir eine Gesamtzeit von $O(m+n\cdot(K+K)) = O(m+n\log C)$ für den Algorithmus von Dijkstra, wie behauptet.

Es bleibt zu zeigen, dass *deleteMin*-Operationen Knoten aus Behälter $B[i]$ in Behälter mit Nummern kleiner als i verschieben.

Lemma 10.7. *Sei i der kleinste Index, für den $B[i]$ nicht leer ist, wobei $i \geq 0$ gilt, und sei min der kleinste Schlüssel in $B[i]$. Dann gilt für alle Schlüssel $x \in B[i]$ die Ungleichung $msd(min,x) < i$.*

Beweis. Sei $x \in B[i]$ beliebig. Der Fall $x = min$ ist leicht, weil $msd(x,x) = -1 < i$ gilt. Für den nichttrivialen Fall $x \neq min$ betrachten wir die Unterfälle $i < K$ und $i = K$. Sei min_o der alte Wert von *min*. Abbildung 10.8 zeigt die Struktur der relevanten Schlüssel.

Fall $i < K$. Der höchstwertige unterscheidende Index von min_o und einem beliebigen $y \in B[i]$ ist i. Das heißt: An Position i in min_o steht eine 0, in x und *min* eine 1, und an allen Positionen mit Index größer als i stimmen min_o, x und *min* überein (α in Abb. 10.8). Daher ist der höchstwertige unterscheidende Index von *min* und x kleiner als i.

Fall $i = K$. Sei $j = msd(min_o, min)$. Dann gilt $j \geq K$, weil $min \in B[K]$. Sei $h = msd(min,x)$. Wir wollen beweisen, dass $h < K$ gilt. Mit α bezeichnen wir die Bits in min_o in Positionen größer als j. Sei A die Zahl, die man aus min_o erhält, wenn man alle Bits in Positionen 0 bis j auf 0 setzt. Die Binärdarstellung von A besteht dann aus α gefolgt von $j+1$ Nullen. Weil min_o an Position j eine 0 stehen hat, *min* dagegen eine 1, gilt $min_o < A + 2^j$ und $A + 2^j \leq min$. Weiter gilt $x \leq min_o + C < A + 2^j + C \leq A + 2^j + 2^K$. Also haben wir

$$A + 2^j \leq min \leq x < A + 2^j + 2^K;$$

daher bestehen die Binärdarstellungen von *min* und x aus α, gefolgt von einer 1, gefolgt von $j-K$ Nullen, gefolgt von einem Bitstring der Länge K. Das bedeutet,

dass die Binärdarstellungen von *min* und x in allen Bits mit Index K und größer übereinstimmen. Daraus folgt $h < K$.

Um die Situation intuitiv besser verständlich zu machen, geben wir noch einen zweiten Beweis für den Fall $i = K$ an. Wir beobachten zunächst, dass die Binärdarstellung von *min* mit α und einer nachfolgenden 1 beginnt. Nun beobachten wir, dass man x erhält, indem man zu min_o eine passende K-Bit-Zahl addiert. Weil $min \leq x$ gilt, muss das letzte Übertragsbit in dieser Binäraddition für Position K entstehen und bis zur Position j weiterlaufen. Andernfalls hätte x an Position j eine 0 stehen und wir hätten $x < min$. Weil in min_o an Position j eine 0 steht, läuft das Übertragsbit aber auch nicht weiter als bis Position j. Daraus ergibt sich, dass die Binärdarstellung von x wie folgt aussieht: Sie beginnt mit α, es folgt eine 1, dann folgen $j-K$ Nullen und schließlich ein beliebiger K-Bit-String. Weil $min \leq x$ gilt, müssen die $j-K$ Nullen auch in der Binärdarstellung von *min* stehen. □

***Aufgabe 10.12.** Man kann Radix-Heaps auch mit Zahldarstellungen bauen, die eine Basis b mit beliebig gewähltem $b \geq 2$ benutzen. In dieser Situation gibt es Behälter $B[i,j]$ für $i = -1,0,1,\ldots,K$ und $0 \leq j \leq b$, wobei $K = 1 + \lfloor \log C / \log b \rfloor$ gilt. Ein erreichter, aber unbearbeiteter Knoten x steht in Behälter $B[i,j]$, wenn $msd(min, d[x]) = i$ gilt und die i-te Ziffer von $d[x]$ gleich j ist. Für jedes i speichern wir auch $|\bigcup_j B[i,j]|$, also die Gesamtanzahl der Knoten, die in den Behältern $B[i,\cdot]$ enthalten sind. Diskutieren Sie die Implementierung der Operationen der Prioritätswarteschlange und zeigen Sie, dass sich ein Kürzeste-Wege-Algorithmus mit Rechenzeit $O(m + n(b + \log C/\log b))$ ergibt. Wie sollte man b wählen, um eine möglichst kleine Schranke zu erhalten?

Wenn die Kantenkosten zufällige natürliche Zahlen im Bereich $0..C$ sind, führt eine kleine Änderung am Algorithmus von Dijkstra mit Radix-Heaps zu linearer erwarteter Zeit [151, 83]. Für jeden Knoten v bezeichnen wir mit $c^{\text{in}}_{\min}(v)$ die minimalen Kosten einer Kante mit Zielknoten v. Wir teilen die Prioritätswarteschlange Q in zwei Teile, eine Menge F mit unbearbeiteten Knoten, deren Schätzdistanz garantiert gleich dem wirklichen Abstand von s ist, und eine Menge B, die alle anderen unbearbeiteten Knoten enthält. Die Menge B wird als Radix-Heap organisiert. Weiter wird ein Wert *min* verwaltet. Knoten werden folgendermaßen bearbeitet.

Falls F nicht leer ist, wird ein beliebiger Knoten aus F entnommen und von ihm ausgehenden Kanten werden relaxiert. Falls F leer ist, wird der Knoten mit der kleinsten Schätzdistanz aus B gewählt und entnommen, und *min* wird auf die Schätzdistanz dieses Knotens gesetzt. Nachdem ein solcher Knoten aus B gewählt wurde, werden die Knoten im ersten nichtleeren Behälter $B[i]$ neu verteilt, falls $i \geq 0$ gilt. Im Umverteilungsprozess gibt es eine kleine Änderung. Wenn ein Knoten v verschoben werden soll und $d[v] \leq min + c^{\text{in}}_{\min}(v)$ gilt, verschieben wir v in F. Dies ist gerechtfertigt, da zukünftige Relaxierungen von Kanten mit Zielknoten v den Wert $d[v]$ nicht mehr verringern können und daher feststeht, dass $d[v]$ den korrekten Abstand von s nach v angibt.

Wir nennen diesen Algorithmus ALD (*average-case linear Dijkstra*). Der Algorithmus ALD ist korrekt, da nach wie vor die Gleichheit $d[v] = \mu(v)$ gilt, wenn v bearbeitet wird. Falls v aus F entnommen wird, wurde dies im vorangegangenen

Absatz gezeigt, und falls v aus B entnommen wird, folgt dies aus der Tatsache, dass v die kleinste Schätzdistanz unter allen nichtbearbeiteten, erreichten Knoten hat.

Satz 10.8 *Sei G ein beliebiger Graph und sei c eine zufällige Funktion von E nach $0..C$. Dann löst der Algorithmus ALD das Problem „kürzeste Wege von einem Startknoten aus“ in erwarteter Zeit $O(m+n)$.*

Beweis. Wir müssen nur noch die Rechenzeitschranke beweisen. Dafür modifizieren wir die amortisierte Analyse von gewöhnlichen Radix-Heaps. Wie zuvor liegt ein Knoten v am Anfang in einem Behälter $B[j]$, für ein $j \leq K$. Wenn v gerade in einen neuen Behälter $B[i]$ (also noch nicht nach F) verschoben worden ist, gilt $d[v] \geq min + c^{\text{in}}_{\min}(v) + 1$, und daher erfüllt der Index i des neuen Behälters von v die Ungleichung $i \geq \lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor + 1$. Um alle Kosten für das Verschieben von v von einem Behälter zum anderen abzudecken, genügt es daher, für die Operation *insert*(v) genau $K - (\lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor + 1) + 1 = K - \lfloor \log(c^{\text{in}}_{\min}(v)+1) \rfloor$ Jetons auf das Konto einzuzahlen. Wenn wir dies über alle Knoten aufsummieren, erhalten wir eine Gesamteinzahlung von

$$\sum_v (K - \left\lfloor \log(c^{\text{in}}_{\min}(v)+1) \right\rfloor) \leq n + \sum_v (K - \left\lceil \log(c^{\text{in}}_{\min}(v)+1) \right\rceil) .$$

Wir müssen den Erwartungswert dieser Summe abschätzen. Für jeden Knoten trägt eine eingehende Kante einen Summanden zu dieser Summe bei. Wenn wir die entsprechenden Summanden über *alle* Kanten summieren, wird die Summe höchstens größer, d. h.,

$$\sum_v (K - \left\lceil \log(c^{\text{in}}_{\min}(v)+1) \right\rceil) \leq \sum_e (K - \lceil \log(c(e)+1) \rceil) .$$

Nun ist $K - \lceil \log(c(e)+1) \rceil$ die Anzahl der führenden Nullen in der Binärdarstellung von $c(e)$ mit K Bits. Unsere Kantenkosten sind uniform zufällig in $0..C$, und $K = 1 + \lfloor \log C \rfloor$. Daher gilt $\text{prob}(K - \lceil \log(c(e)+1) \rceil \geq k) = |0..2^{K-k} - 1| / |0..C| = 2^{K-k}/(C+1) \leq 2^{-(k-1)}$. (Die letzte Ungleichung gilt wegen $C \geq 2^{K-1}$.) Mit (A.1) und (A.13) erhalten wir für jede Kante e:

$$\text{E}[K - \lceil \log(c(e)+1) \rceil] = \sum_{k \geq 1} \text{prob}(K - \lceil \log(c(e)+1) \rceil \geq k) \leq \sum_{k \geq 1} 2^{-(k-1)} = 2.$$

Mit der Linearität des Erwartungswerts ergibt sich daraus:

$$\text{E}\left[\sum_e (K - \lceil \log(c(e)+1) \rceil) \right] = \sum_e \text{E}[K - \lceil \log(c(e)+1) \rceil] \leq \sum_e 2 = O(m) .$$

Daher sind die gesamten erwarteten Kosten der *deleteMin*-Operationen $O(m+n)$. Die Zeit für die *decreaseKey*-Operationen ist $O(m)$, die Zeit für alle anderen Operationen zusammen ebenfalls $O(m+n)$. □

Man beachte, dass für den Beweis von Satz 10.8 nicht benötigt wird, dass die Kantenkosten unabhängig sind. Es genügt, wenn für jede einzelne Kante e der Wert $c(e)$ uniform in $0..C$ gewählt wird.

Es sieht etwas seltsam aus, dass der Wert C der maximal zulässigen Kantenkosten in Satz 10.8 in der Voraussetzung, aber nicht in der Aussage erscheint. Tatsächlich kann man zeigen, dass ein ähnliches Ergebnis für reellwertige Kantenkosten gilt.

****Aufgabe 10.13.** Erklären Sie, wie man Algorithmus ALD modifizieren kann, damit auch Eingaben bearbeitet werden können, bei denen c eine Zufallsfunktion von E mit Werten im reellen Intervall $(0,1]$ ist. Die erwartete Zeit sollte nach wie vor $O(m+n)$ sein. Welche Annahmen über die Darstellung von Kantenkosten und über die verfügbaren Maschinenbefehle werden benötigt? *Hinweis*: Lösen Sie zuerst Aufgabe 10.11. Der engste Behälter sollte Weite $\min_{e\in E} c(e)$ haben. Nachfolgende Behälter haben geometrisch wachsende Weiten.

10.6 Beliebige Kantenkosten (Der Algorithmus von Bellman und Ford)

Bei azyklischen Graphen und bei nichtnegativen Kantenkosten ist es uns gelungen, mit m Kantenrelaxierungen auszukommen. Für beliebige Graphen und beliebige Kantenkosten ist kein solches Ergebnis bekannt. Jedoch lässt sich leicht einsehen, dass $O(nm)$ Kantenrelaxierungen ausreichen, um zu garantieren, dass das Korrektheitskriterium von Lemma 10.3 erfüllt ist: Der Algorithmus von Bellman und Ford [20, 69], der in Abb. 10.9 dargestellt ist, führt $n-1$ Runden durch. In jeder Runde wird jede Kante einmal relaxiert. Weil einfache Wege aus höchstens $n-1$ Kanten bestehen, ist jeder kürzeste Weg ohne Knotenwiederholung eine Teilfolge dieser Relaxierungen. Nach Lemma 10.1 und Lemma 10.3 gilt daher nach der Ausführung dieser Runden $d[v] = \mu(v)$ für alle Knoten v mit $-\infty < \mu(v) < \infty$. Zudem kodiert das Array *parent* kürzeste Wege von s zu diesen Knoten. Für Knoten v, die von s aus nicht erreichbar sind, gilt $d[v] = \infty$, wie gewünscht.

Es ist nicht ganz so offensichtlich, wie man die Knoten v identifizieren kann, für die $\mu(v) = -\infty$ gilt. Nach Übungsaufgabe 10.1 sind dies die Knoten, die von s aus auf einem Weg erreichbar sind, der einen Kreis mit negativer Länge enthält. Entscheidend ist die folgende Eigenschaft negativer Kreise:

Beh. 1: Wenn $C = \langle v_0, v_1, \ldots, v_k\rangle$ mit $v_0 = v_k$ ein Kreis mit negativer Länge in G ist, der von s aus erreichbar ist, dann gibt es ein i in $1..k$ mit $d[v_{i-1}] + c((v_{i-1}, v_i)) < d[v_i]$.

Beweis: Weil C von s aus erreichbar ist, gilt $d[v_i] < +\infty$ für alle i in $1..k$. Angenommen, für alle diese i gilt $d[v_{i-1}] + c((v_{i-1}, v_i)) \geq d[v_i]$. Wenn wir diese k Ungleichungen addieren, heben sich die $d[v_i]$-Terme gegenseitig auf, und wir erhalten die Ungleichung $\sum_{1\leq i\leq k} c((v_{i-1}, v_i)) \geq 0$. Diese bedeutet, dass C nichtnegative Länge hat.

Beh. 2: Sei v von s aus erreichbar. Dann gilt $\mu(v) = -\infty$ genau dann wenn es eine Kante $e = (u,w)$ gibt, so dass $d[u] + c(e) < d[w] < +\infty$ gilt und v von w aus erreichbar ist.

```
Function BellmanFord(s : NodeId) : NodeArray×NodeArray
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ ∪ {−∞,∞}          // Abstand vom Startknoten
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    d[s] := 0;   parent[s] := s                      // Schleife kennzeichnet Wurzel
    for i := 1 to n − 1 do
        forall e ∈ E do relax(e)                     // Runde i
    forall e = (u,v) ∈ E do                          // Nachbearbeitung
        if d[u] + c(e) < d[v] then infect(v)
    return (d, parent)

Procedure infect(v)
    if d[v] > −∞ then
        d[v] := −∞
        foreach (v,w) ∈ E do infect(w)
```

Abb. 10.9. Der Algorithmus von Bellman und Ford für kürzeste Wege in beliebigen Graphen

Beweis: Sei zunächst $\mu(v) = -\infty$. Dann ist v von s aus über einen Kreis $C = \langle v_0, v_1, \ldots, v_k \rangle$ mit negativer Länge erreichbar. Nach Beh. 1 enthält dieser Kreis eine Kante $e = (v_{i-1}, v_i)$ mit $d[v_{i-1}] + c((v_{i-1}, v_i)) < d[v_i]$. Zudem ist v von v_i aus erreichbar. Sei nun $e = (u,w)$ eine Kante mit $d[u] + c(e) < d[w] < +\infty$. Wir überlegen zunächst, dass $\mu(w) = -\infty$ gelten muss. Andernfalls hätten wir auch $-\infty < \mu(u) < +\infty$ und damit nach Lemma 10.3 $\mu(u) = d[u]$ und $\mu(w) = d[w]$. Die Ungleichung $\mu(u) + c(e) < \mu(w)$ kann aber nicht gelten. Da also $\mu(w) = -\infty$ gilt, ist w von s aus über einen Kreis mit negativer Länge erreichbar. Dies gilt dann auch für v.
Es ergibt sich folgender Ansatz, um in einer Nachbearbeitungsrunde die Knoten v mit $\mu(v) = -\infty$ zu identifizieren (und $d[v]$ auf $-\infty$ zu setzen): Für jede Kante $e = (u,v)$ wird einmal getestet, ob $d[u] + c(e) < d[v]$ gilt. Wenn dies so ist, erklärt man v für „infiziert“, setzt $d[v] := -\infty$ und ruft eine rekursive Prozedur *infect*(v) auf, die analog zur Tiefensuche bewirkt, dass die d-Werte aller von v aus erreichbaren Knoten w auf $-\infty$ gesetzt werden, sofern sie nicht schon vorher diesen Wert hatten.

Aufgabe 10.14. Zeigen Sie, dass in der Nachbearbeitung es genau für die Knoten mit $\mu(v) = -\infty$ einen oder mehrere Aufrufe *infect*(v) gibt, und dass die gesamte Nachbearbeitung in Zeit $O(m)$ abläuft. *Hinweis*: Argumentieren Sie wie bei der Analyse der Tiefensuche.

Aufgabe 10.15. Jemand schlägt ein alternatives Nachbearbeitungsverfahren vor: Setze $d[v]$ auf $-\infty$ für alle Knoten v mit $d[v] < +\infty$, für die der durch die Vorgängerzeiger *parent* angegebene Weg nicht zum Startknoten s führt. Geben Sie ein Beispiel an, bei dem diese Methode einen Knoten v mit $\mu(v) = -\infty$ übersieht.

Aufgabe 10.16 (Arbitrage). Betrachten Sie eine Menge C von Währungen mit Wechselkurs r_{ij} zwischen Währungen i und j (d. h., man erhält r_{ij} Einheiten von Währung j für eine Einheit von Währung i). Eine *Währungsarbitrage* ist möglich, wenn es eine Folge von elementaren Umtauschvorgängen (*Transaktionen*) gibt, die

mit einer Einheit in einer Währung beginnt und mit mehr als einer Einheit derselben Währung endet. (a) Erklären Sie, wie man ermitteln kann, ob eine Matrix von Wechselkursen Währungsarbitrage zulässt. *Hinweis*: $\log(xy) = \log x + \log y$. (b) Verfeinern Sie Ihren Algorithmus dahingehend, dass er eine Folge von Umtauschtransaktionen ermittelt, die den durchschnittlichen Gewinn *pro Transaktion* maximiert!

In Abschnitt 10.10 werden weitere Verfeinerungen für den Algorithmus von Bellman und Ford beschrieben, die für ein gutes Verhalten in der Praxis benötigt werden.

10.7 Kürzeste Wege zwischen allen Knotenpaaren und Knotenpotenziale

Das Problem, in allgemeinen Graphen mit beliebigen Kantenkosten für alle Knotenpaare Kürzeste-Wege-Distanzen zu berechnen (*all-pairs*), ist gleichbedeutend mit n Kürzeste-Wege-Berechnungen von einem Startknoten aus (*single-source*) und kann daher in Zeit $O(n^2m)$ gelöst werden. Jedoch ist hier eine erhebliche Verbesserung möglich. Wir werden zeigen, dass es ausreicht, ein allgemeines Problem mit einem Startknoten und zusätzlich n Probleme mit einem Startknoten in Graphen mit nichtnegativen Kantenkosten zu lösen. Auf diese Weise erreichen wir eine Rechenzeit von $O(nm + n(m + n\log n)) = O(nm + n^2 \log n)$. Wir benötigen hierfür den Begriff von Knotenpotenzialen.

Eine *(Knoten-)Potenzialfunktion* ordnet jedem Knoten v eine Zahl $pot(v)$ zu. Die *reduzierten Kosten* $\bar{c}(e)$ einer Kante $e = (v,w)$ werden dann als

$$\bar{c}(e) = pot(v) + c(e) - pot(w)$$

definiert.

Lemma 10.9. *Seien p und q Wege von v nach w. Dann gilt $\bar{c}(p) = pot(v) + c(p) - pot(w)$, und $\bar{c}(p) \leq \bar{c}(q)$ gilt genau dann wenn $c(p) \leq c(q)$. Insbesondere sind die kürzesten Wege bezüglich $\bar{c}$ genau dieselben wie die bezüglich c.*

Beweis. Die zweite und die dritte Feststellung folgen unmittelbar aus der ersten. Um die erste Behauptung zu beweisen, betrachte $p = \langle e_1, \ldots, e_k \rangle$ mit $e_i = (v_{i-1}, v_i)$, $v = v_0$ und $w = v_k$. Dann gilt

$$\begin{aligned} \bar{c}(p) &= \sum_{1 \leq i \leq k} \bar{c}(e_i) \;=\; \sum_{1 \leq i \leq k} (pot(v_{i-1}) + c(e_i) - pot(v_i)) \\ &= pot(v_0) + \sum_{1 \leq i \leq k} c(e_i) - pot(v_k) = pot(v_0) + c(p) - pot(v_k)\,. \end{aligned}$$ □

Aufgabe 10.17. Mit Hilfe von Knotenpotenzialen lassen sich Graphen erzeugen, die negative Kantenkosten, aber keine negativen Kreise haben: Erzeuge einen (zufälligen) Graphen, weise jeder Kante e (zufällig gewählte) *nichtnegative* Kosten $c(e)$ zu, weise jedem Knoten einen (zufälligen) Potenzialwert $pot(v)$ zu, und setze die Kosten von $e = (u,v)$ auf $\bar{c}(e) = pot(u) + c(e) - pot(v)$. Zeigen Sie, dass dieses Vorgehen nie negative Kreise erzeugt.

Kürzeste Wege zwischen allen Knotenpaaren, ohne negative Kreise
füge neuen Knoten s hinzu
füge Kanten (s,v) mit Kosten 0 hinzu, für $v \in V$ // keine neuen Kreise, Zeit $O(n)$
berechne $\mu(v)$ für alle v mit Bellman-Ford // Zeit $O(nm)$
setze $pot(v) = \mu(v)$; berechne reduzierte Kosten $\bar{c}(e)$, für $e \in E$ // Zeit $O(m)$
forall Knoten x **do** // Zeit $O(n(m+n\log n))$
    benutze Dijkstra-Algorithmus, um reduzierte Kürzeste-Wege-Distanzen $\bar{\mu}(x,v)$
    zu berechnen, mit Startknoten x und reduzierten Kantenkosten $\bar{c}$
// übersetze Distanzen zurück in die ursprüngliche Kostenfunktion // Zeit $O(n^2)$
forall $e = (v,w) \in V \times V$ **do** $\mu(v,w) := \bar{\mu}(v,w) + pot(w) - pot(v)$ // nach Lemma 10.9

Abb. 10.10. Algorithmus für kürzeste Wege für alle Knotenpaaren, ohne negative Kreise

Lemma 10.10. *Angenommen, G hat keine negativen Kreise, und alle Knoten sind von s aus erreichbar. Wenn für jedes $v \in V$ der Wert $pot(v) = \mu(v)$ als Knotenpotenzial gewählt wird, dann sind die reduzierten Kantenkosten alle nichtnegativ.*

Beweis. Weil alle Knoten von s aus erreichbar sind und es keine negativen Kreise gibt, gilt $\mu(v) \in \mathbb{R}$ für alle v. Daher sind die Knotenpotenziale und die reduzierten Kosten wohldefiniert. Für eine beliebige Kante $e = (v,w)$ gilt $\mu(v) + c(e) \geq \mu(w)$, also $\bar{c}(e) = \mu(v) + c(e) - \mu(w) \geq 0$. □

Satz 10.11 *Das Problem, Kürzeste-Wege-Distanzen für alle Knotenpaare zu berechnen, lässt sich für einen Graphen G ohne negative Kreise in Zeit $O(nm + n^2 \log n)$ lösen.*

Beweis. Der Algorithmus ist in Abb. 10.10 angegeben. Wir fügen zu G einen Hilfsknoten s und Kanten (s,v) mit Kosten 0 hinzu, für alle Knoten v in G. Dies erzeugt keine negativen Kreise und lässt $\mu(v,w)$ für die Knoten im ursprünglichen Graphen unverändert. Dann lösen wir das Kürzeste-Wege-Problem für Startknoten s und setzen $pot(v) = \mu(v)$ für jeden Knoten v. Nun berechnen wir die reduzierten Kantenkosten und lösen dann mit dem Algorithmus von Dijkstra in G das Kürzeste-Wege-Problem von einem Startknoten aus, und zwar für jeden Knoten x als Startknoten. Schließlich übersetzen wir die ermittelten reduzierten Distanzen wieder zurück in die ursprüngliche Kostenfunktion. Die Berechnung der Potenzialwerte mit dem Algorithmus von Bellman und Ford kostet Zeit $O(nm)$; die n Aufrufe des Algorithmus von Dijkstra nehmen Zeit $O(n(m+n\log n))$ in Anspruch. Vorbereitung und Nachverarbeitung kosten Zeit $O(n^2)$. □

Die Annahme, dass G keine negativen Kreise hat, kann weggelassen werden [145].

Aufgabe 10.18. Der *Durchmesser D* eines Graphen G ist als der maximale Abstand zwischen irgendwelchen Paaren von Knoten definiert. Der Durchmesser lässt sich natürlich leicht bestimmen, wenn man die Kürzeste-Wege-Distanzen zwischen allen

Knotenpaaren berechnet hat. Hier wollen wir überlegen, wie sich mit nur einer konstanten Anzahl von Kürzeste-Wege-Berechnungen von einem Startknoten aus der Durchmesser eines stark zusammenhängenden Graphen G *approximieren* lässt. (a) Für einen beliebigen Startknoten s definieren wir $D'(s) := \max_{u \in V} \mu(u)$. Zeigen Sie, dass für ungerichtete Graphen die Ungleichung $D'(s) \leq D \leq 2D'(s)$ gilt. Zeigen Sie auch, dass es bei gerichteten Graphen keine entsprechende Beziehung gibt. Nun setzen wir $D''(s) := \max_{u \in V} \mu(u,s)$. Zeigen Sie, dass sowohl in ungerichteten als auch in gerichteten Graphen die Ungleichung $\max\{D'(s), D''(s)\} \leq D \leq D'(s) + D''(s)$ gilt. (b) Wie sollte ein Graph dargestellt sein, damit die Berechnung kürzester Wege sowohl mit Startknoten s als auch mit Zielknoten s effizient vonstatten geht? (c) Kann man die Approximationsgüte verbessern, indem man mehr als einen Knoten s betrachtet?

10.8 Kürzeste-Wege-Anfragen

Oft interessieren wir uns für einen kürzesten Weg von einem gegebenen Startknoten s zu einem gegebenen Zielknoten t; Routenplanung in einem Verkehrsnetzwerk ist ein solches Szenario. Wir erklären hier einige Techniken, die es gestatten, solche *Kürzeste-Wege-Anfragen* effizient auszuführen, und begründen, weshalb sie für die Anwendung bei der Routenplanung nützlich sind. In diesem Abschnitt gehen wir immer von nichtnegativen Kantenkosten aus.

Wir beginnen mit einer Technik, die *frühes Stoppen* heißt. Dabei lässt man den Algorithmus von Dijkstra mit Startknoten s laufen, bricht die Suche aber ab, sobald t aus der Prioritätswarteschlange entnommen wird, weil zu diesem Zeitpunkt die Kürzeste-Wege-Distanz von t bekannt ist. Das verkürzt die Rechenzeit, außer wenn t unglücklicherweise der Knoten mit der maximalen Distanz von s ist. Unter der Annahme, dass in einer Anwendung jeder Knoten mit gleicher Wahrscheinlichkeit als Zielknoten auftritt, halbiert frühes Stoppen im Mittel die Anzahl der bearbeiteten Knoten. In der Praxis der Routenplanung spart frühes Stoppen viel mehr, weil moderne Navigationssysteme für Autos die Karte eines ganzen Kontinents enthalten, aber meist nur für Fahrten von bis zu wenigen hundert Kilometern benutzt werden.

Eine weitere einfache und allgemeine Heuristik ist *bidirektionale Suche* von s aus vorwärts und von t aus rückwärts, bis die Suchfronten aufeinandertreffen. Genauer gesagt lässt man dabei zwei Kopien des Algorithmus von Dijkstra nebeneinander ablaufen, die eine von s startend, die andere von t (und auf dem Umkehrgraphen arbeitend). Die beiden Kopien haben jeweils eigene Prioritätswarteschlangen, etwa Q_s und Q_t. Man lässt dabei die Suchbereiche ungefähr im selben Tempo wachsen, z. B. indem man einen Knoten aus Q_s entnimmt, wenn $\min Q_s \leq \min Q_t$ gilt, sonst einen Knoten aus Q_t.

Die Versuchung liegt nahe, das Verfahren abzubrechen, sobald ein Knoten u aus beiden Prioritätswarteschlangen entnommen worden ist, und zu behaupten, dass $\mu(t) = \mu(s,u) + \mu(u,t)$ gilt. (Die Distanz $\mu(u,t)$ wird ja von der Version des Algorithmus von Dijkstra ermittelt, die ausgehend von t auf dem Umkehrgraphen abläuft.) Dies ist nicht ganz korrekt, aber fast.

Aufgabe 10.19. Geben Sie ein Beispiel an, in dem *u nicht* auf einem kürzesten Weg von s nach t liegt.

Es ist jedoch tatsächlich so, dass genug Information vorliegt, um einen kürzesten Weg von s nach t zu bestimmen, sobald ein Knoten u aus beiden Prioritätswarteschlangen entnommen worden ist und die beiden Kopien des Dijkstra-Algorithmus angehalten worden sind. Seien d_s bzw. d_t die Schätzdistanzen zu den Kopien mit Startknoten s bzw. t zu diesem Zeitpunkt. Wir wollen zeigen, dass aus $\mu(t) < \mu(s,u) + \mu(u,t)$ folgt, dass es in Q_s einen Knoten v mit $\mu(t) = d_s[v] + d_t[v]$ gibt.

Sei dazu $p = \langle s = v_0, \ldots, v_{i-1}, v_i, \ldots, v_k = t \rangle$ ein kürzester Weg von s nach t. Wir wählen das größte i mit der Eigenschaft, dass v_{i-1} aus Q_s entnommen und bearbeitet worden ist, wenn die Algorithmen angehalten werden. Dann gilt in diesem Moment $d_s[v_i] = \mu(s,v_i)$ und $v_i \in Q_s$. Weiterhin gilt $\mu(s,u) \leq \mu(s,v_i)$, weil u schon bearbeitet worden ist, nicht aber v_i. Nun beobachten wir, dass

$$\mu(s,v_i) + \mu(v_i,t) = c(p) < \mu(s,u) + \mu(u,t) \,,$$

gilt: Die Gleichung gilt, weil p ein kürzester Weg von s nach t ist, die Ungleichung nach der Annahme über u. Wir subtrahieren $\mu(s,v_i)$ von beiden Seiten und erhalten:

$$\mu(v_i,t) < \underbrace{\mu(s,u) - \mu(s,v_i)}_{\leq 0} + \mu(u,t) \leq \mu(u,t) \,.$$

Weil u in der Suche von t aus schon bearbeitet worden ist, muss dies auch für v_i der Fall sein, d. h., es gilt $d_t[v_i] = \mu(v_i,t)$, wenn die Algorithmen angehalten werden.

Wir erhalten also die kürzeste-Wege-Distanz von s nach t, indem wir nicht nur den ersten Knoten betrachten, der in beiden Algorithmenkopien bearbeitet wird, sondern zudem alle Knoten etwa in Q_s. Wir durchlaufen alle diese Knoten v und bestimmen die kleinste Summe $d_s[v] + d_t[v]$.

Der Algorithmus von Dijkstra bearbeitet Knoten in der Reihenfolge wachsender Abstände vom Startknoten. In anderen Worten, er baut einen immer größer werdenden Kreis, dessen Mittelpunkt der Startknoten ist. Der Kreis wird durch die Kürzeste-Wege-Metrik im Graphen definiert. In der Anwendung des Kürzeste-Wege-Problems in der Routenplanung liegt es recht nahe, geometrische Kreise mit dem Startknoten als Mittelpunkt zu betrachten und anzunehmen, dass Kürzeste-Wege-Kreise und geometrische Kreise in etwa gleich sind. Wir können dann die durch bidirektionale Suche erzielte Beschleunigung mit folgender heuristischer Überlegung schätzen: Ein Kreis mit einem bestimmten Radius hat eine doppelt so große Fläche wie zwei Kreise mit dem halben Radius. Wir könnten daher hoffen, dass die bidirektionale Suche etwa um den Faktor 2 schneller zum Ziel kommt als der Algorithmus von Dijkstra selbst („unidirektionale Suche“).

Aufgabe 10.20 (Bidirektionale Suche). (a) Betrachten Sie bidirektionale Suche in einem Gittergraphen, in dem alle Kanten Gewicht 1 haben. Um welchen Faktor geht dies schneller als unidirektionale Suche? *(b) Versuchen Sie, eine Familie von Graphen zu finden, bei der die bidirektionale Suche im Mittel (über alle möglichen Paare

von Start- und Zielknoten) polynomiell weniger Knoten besucht als unidirektionale Suche. *Hinweis*: Betrachten Sie Zufallsgraphen mit $O(n)$ Kanten oder Hypercubes. (c) Geben Sie ein Beispiel an, wo bidirektionale Suche in einem konkreten Straßennetzwerk *länger* dauert als unidirektionale Suche. *Hinweis*: Betrachten Sie eine dicht besiedelte Stadt mit dünn besiedelter Umgebung. (d) Entwickeln Sie eine Strategie für bidirektionale Suche, so zwischen der Suche in Vorwärtsrichtung (von s aus) und der Suche in Rückwärtsrichtung (von t aus) hin- und herzuschalten, dass nie mehr als doppelt so viele Knoten bearbeitet werden als bei einer unidirektionalen Suche.

Im Folgenden werden noch drei etwas kompliziertere Techniken beschrieben, die zudem in weniger Situationen anwendbar sind. Wenn sie aber anwendbar sind, führen sie normalerweise zu größeren Einsparungen. Alle diese Techniken ahmen das Verhalten von Menschen bei der Routenplanung nach.

10.8.1 Zielgerichtete Suche

Die Grundidee ist hier, den Suchraum so „umzugewichten“, dass der Algorithmus von Dijkstra nicht eine immer größer werdende Kreisscheibe aufbaut, sondern ein Gebiet, das sich stärker in Richtung des Zielknotens ausdehnt, wie in Abb. 10.11 schematisch dargestellt. Wir nehmen dazu an, dass eine Funktion $f\colon V \to \mathbb{R}$ vorliegt, die den Abstand zum Zielknoten schätzt, d. h., dass für jeden Knoten v die Zahl $f(v)$ ein Schätzwert für $\mu(v,t)$ ist. Diese Schätzwerte werden benutzt, um die Distanzfunktion zu modifizieren. Für jede Kante $e = (u,v)$ setzen wir[3]

$$\overline{c}(e) = c(e) + f(v) - f(u) \; .$$

Wir lassen den Algorithmus von Dijkstra mit den modifizierten Kosten ablaufen. (Hierzu nehmen wir vorläufig einfach an, dass die modifizierten Kantenkosten nichtnegativ sind. Details hierzu folgen gleich.) Dieser Ablauf findet kürzeste Wege bezüglich $\overline{c}$. Nach Lemma 10.9 sind dies aber ebenfalls kürzeste Wege bezüglich der ursprünglichen Kosten c. Der Unterschied ist, dass die Knoten eventuell in einer anderen Reihenfolge aus der Prioritätswarteschlange entnommen und bearbeitet werden als bei der Verwendung der Kosten c. Wir hoffen aber, dass aufgrund der modifizierten Kosten zum Erreichen von t weniger Knoten bearbeitet werden müssen.

Wenn man eine beliebige Folge R von Relaxierungen (wie in Abschnitt 10.1) bezüglich $\overline{c}$ bzw. c parallel ablaufen lässt, dann besteht nach jedem Schritt zwischen den jeweiligen Schätzdistanzen $\overline{d}$ und d die Beziehung $\overline{d}[v] = d[v] + f(v) - f(s)$, wie sich durch Induktion über die Relaxierungen leicht zeigen lässt. Dies gilt natürlich auch für die Relaxierungsreihenfolge, die durch den Algorithmus von Dijkstra mit $\overline{c}$ erzeugt wird. Der konstante Summand $f(s)$ ist offensichtlich irrelevant. Das

[3] In Abschnitt 10.7 hatten wir bei Kante (u,v) den Potenzialwert des Startknotens u addiert und den Potenzialwert des Zielknotens v subtrahiert. Hier geschieht genau das Umgekehrte. Technisch wird also $-f$ als Potenzialfunktion benutzt. Der Grund für diese Vorzeichenänderung ist, dass wir in Lemma 10.10 den Wert $\mu(s,v)$ als das Knotenpotenzial benutzen wollten, hier aber $f(v)$ den Wert $\mu(v,t)$ schätzt, mit Orientierung auf den Zielknoten t hin.

beschriebene Verfahren entspricht also dem Algorithmus von Dijkstra mit der Modifikation, dass für die Anordnung der Knoten in der Prioritätswarteschlange nicht $d[v]$, sondern $d[v] + f(v)$ benutzt wird, und in jeder Iteration ein Knoten v bearbeitet wird, der $d[v] + f(v)$ minimiert. Das beschriebene Verfahren ist unter dem Namen *A*-Suche* bekannt. In diesem Zusammenhang heißt f auch oft *Heuristik* oder *heuristische Schätzfunktion*.

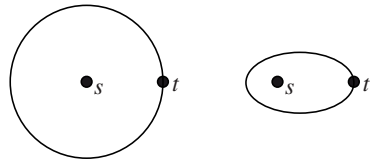


Abb. 10.11. Die gewöhnliche Dijkstra-Suche baut einen immer größer werdenden kreisförmigen Bereich mit dem Startknoten s als Mittelpunkt; zielgerichtete Suche baut einen Bereich, der sich dem Zielknoten t zuneigt.

Bevor wir Anforderungen an die Schätzfunktion f formulieren, betrachten wir ein spezifisches Beispiel. Nehmen wir als Gedankenexperiment an, dass für alle Knoten v die Gleichung $f(v) = \mu(v,t)$ gilt. Dann gilt $\overline{c}(e) = c(e) + \mu(v,t) - \mu(u,t)$ für jede Kante $e = (u,v)$, und daher haben alle Kanten auf kürzesten Wegen von s nach t modifizierte Kosten 0. Wenn $e = (u,v)$ eine Kante ist, für die u auf einem solchen kürzesten Weg liegt, die Kante e jedoch nicht, hat e positive modifizierte Kosten. Daher arbeitet der Algorithmus von Dijkstra mit den modifizierten Kosten nur auf kürzesten Wegen von s nach t, ohne nach rechts oder links zu sehen, bis t bearbeitet wird und der Algorithmus endet.

Damit A*-Suche ausgeführt werden kann, muss auf den Graphen mit Kantenkosten $\overline{c}$ der Algorithmus von Dijkstra anwendbar sein. Dies führt zu folgender Anforderung an die Schätzfunktion f: Für jede Kante $e = (u,v)$ muss die Ungleichung $\overline{c}(e) \geq 0$ gelten, also $c(e) + f(v) \geq f(u)$ sein. In Worten: Die Schätzung für den Abstand von u nach t darf nicht größer sein als die Schätzung für den Abstand von v nach t plus die Kosten $c(u,v)$ für den Schritt von u nach v. Diese Eigenschaft nennt man *Konsistenz der Schätzwerte* $f(v)$. Sie hat eine interessante Konsequenz. Wenn $p = \langle v = v_0, v_1, \ldots, v_k = t \rangle$ ein beliebiger Weg von v nach t ist, dann ergibt die Addition der Konsistenzrelationen $c((v_{i-1}, v_i)) + f(v_i) \geq f(v_{i-1})$ über $1 \leq i \leq k$ die Ungleichung $c(p) + f(t) \geq f(v)$. Daraus folgt $\mu(v,t) \geq f(v) - f(t)$. Wir treffen die naheliegende Festlegung, dass der Schätzwert $f(t)$ für $\mu(t,t) = 0$ selbst gleich 0 ist (falls nötig, kann man von allen Schätzwerten $f(v)$ den Wert $f(t)$ subtrahieren). Dann ergibt sich, dass $f(v)$ eine untere Schranke für $\mu(v,t)$ ist.

Was ist im Fall der Routenplanung in einem Straßennetzwerk eine gute heuristische Schätzfunktion? Routenplaner lassen den Benutzer oft zwischen *kürzesten* und *schnellsten* Verbindungen wählen. Im Fall von kürzesten Verbindungen ist eine naheliegende und einfache untere Schranke $f(v)$ die Länge der geraden Strecke zwischen v und t, also der geometrische Abstand. In der Literatur wird berichtet, dass die Re-

chenzeiten mit diesem Ansatz etwa um den Faktor 4 schneller werden. Für schnellste Wege könnten wir den Quotienten aus geometrischem Abstand und der Geschwindigkeit wählen, die für die beste Straßenart angenommen wird. Diese Schätzung ist natürlich sehr optimistisch, weil Fahrtziele sich oft inmitten einer Stadt befinden. Für diesen Ansatz wurden daher auch keine großen Rechenzeitverbesserungen berichtet. Es gibt auch ausgefeiltere Methoden für die Berechnung von unteren Schranken; für eine detaillierte Diskussion sei der auf die Arbeit [84] verwiesen.

10.8.2 Hierarchien

Straßennetze bestehen gewöhnlich aus einer Hierarchie von Straßen, in Deutschland etwa Autobahnen und autobahnähnliche Straßen, Bundesstraßen, Landesstraßen, Gemeindeverbindungen, innerstädtische Straßen. Normalerweise wird auf den Straßen mit höherem Status eine höhere durchschnittliche Geschwindigkeit erreicht, und daher folgen schnellste Verbindungswege oft dem Muster, dass man auf Straßen mit niedrigem Status beginnt, mehrere Male auf Straßen mit immer höherem Status wechselt, den größten Teil der Strecke auf einer Straße mit hohem Status fährt, und dann in der Nähe des Ziels wieder auf Straßen mit niedrigerem Status wechselt. Ein heuristischer Ansatz wäre daher, die Suche auf Straßen mit höherem Status zu beschränken, mit Ausnahme passend festgelegter Umgebungen von Start und Ziel. Man beachte aber, dass die Wahl dieser Nachbarschaften nicht offensichtlich ist, und dass mit einer solchen Heuristik die Optimalität geopfert wird. Vielleicht fällt dem Leser aus eigener Erfahrung ein Beispiel ein, wo für die schnellste Verbindung auch weitab von Start und Ziel Abkürzungen über kleinere Straßen benutzt werden müssen. Korrektheit lässt sich mit der Idee von Straßenhierarchien verbinden, wenn die Hierarchie algorithmisch definiert wird und nicht aus der offiziellen Klassifikation der Straßen übernommen wird. Wir skizzieren nun einen solchen Ansatz [180], genannt *Schnellstraßenhierarchien* (engl.: *highway hierarchies*). Dabei wird zunächst ein Konzept „Nahbereich“ definiert, z. B. alles im Abstand von bis zu 10 km von Start oder Ziel. Eine Kante $(u,v) \in E$ ist eine *Schnellstraßenkante* bezüglich dieses Nahbereichskonzepts, wenn es einen Startknoten s und einen Zielknoten t gibt, so dass (u,v) auf einem schnellsten Weg von s nach t liegt, aber weder im Nahbereich von s noch im Nahbereich von t, d. h., weder von s aus innerhalb des festgelegten Nahbereichsradius erreichbar ist noch von t aus (im Umkehrgraphen). Das Netzwerk, das sich so ergibt, heißt das *Schnellstraßennetz*. In ihm kommen normalerweise viele Knoten mit Grad 2 vor: Wenn eine langsame Straße in eine schnelle Straße einmündet, wird die langsame Straße meist nicht auf einer schnellsten Route liegen, wenn die Einmündung weder zum Nachbereich des Startpunkts noch zu dem des Zielpunkts gehört. Daher wird diese Straße nicht im Schnellstraßennetz vorkommen. Die Einmündung hat also Grad 3 im ursprünglichen Straßennetz, aber Grad 2 im Schnellstraßennetz. Zwei Kanten, die an einem Knoten mit Grad 2 aneinanderstoßen, können zu einer einzigen Kante zusammengezogen werden. Auf diese Weise ergibt sich der *Kern* des Schnellstraßennetzes. Wenn diese Prozedur (definiere Nahbereiche, finde Schnellstraßenkanten, kontrahiere Kantenpaare an Knoten mit Grad 2) iteriert, erhält man eine Hierarchie von Schnellstraßennetzen. Beispielsweise

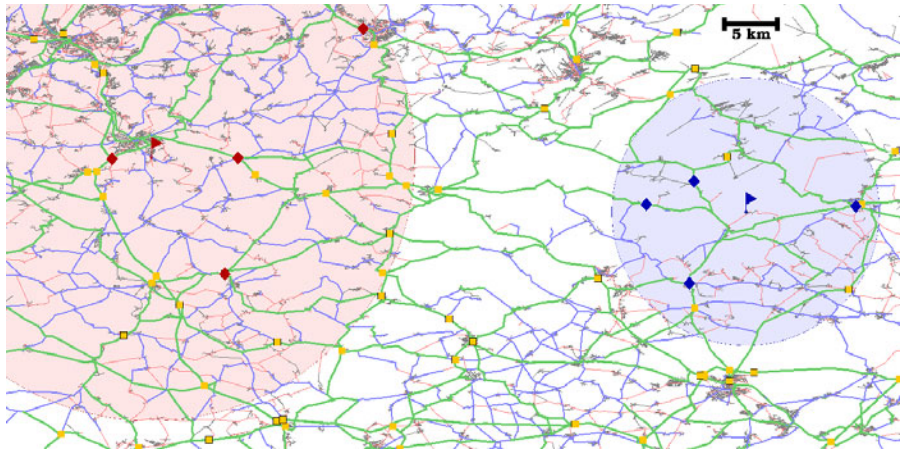


Abb. 10.12. Eine schnellste Route zwischen zwei Punkten (den Fähnchen) irgendwo zwischen Saarbrücken und Karlsruhe zu finden läuft darauf hinaus, die 2×4 entsprechenden *Zugangsknoten* (Karos) zu ermitteln, die 16 Distanzen zwischen allen Paaren der Zugangsknoten aus einer Tabelle auszulesen und zu überprüfen, dass die beiden Kreisscheiben, die den *Nachbarschaftsfilter* definieren, sich nicht überschneiden. Die kleinen Quadrate stehen für weitere Übergangsknoten.

ergaben sich in Versuchen mit den Straßennetzen von Europa und von Nordamerika Hierarchien mit bis zu 10 Ebenen. Die Berechnung von schnellsten Routen auf der Basis der sich ergebenden Schnellstraßenhierarchie kann mehrere tausend Male schneller sein als die einfache Benutzung des Algorithmus von Dijkstra.

10.8.3 Routing mit Übergangsknoten

Mit Hilfe einer anderen Beobachtung aus dem täglichen Leben können wir sogar noch schnellere Routingverfahren erhalten [17]. Wenn man „weit weg“ fährt, wird man seine lokale Umgebung über eine von wenigen „wichtigen“ Straßenkreuzungen oder -anschlussstellen verlassen. Es stellt sich heraus, dass in wirklichen Straßennetzwerken etwa 99% aller schnellsten Routen durch etwa $O(\sqrt{n})$ *Übergangsknoten* führen, die vorberechnet werden können, z. B. unter Benutzung von Schnellstraßenhierarchien. Zudem gehen für jeden als Start oder Ziel gewählten Knoten alle Fernverbindungen über einen von etwa zehn dieser Übergangsknoten – die *Zugangsknoten*. In einer Vorverarbeitungsrunde berechnen wir eine vollständige Distanztabelle für alle Paare von Übergangsknoten und von allen Knoten zu ihren Zugangsknoten. Weiter nehmen wir an, dass es eine Möglichkeit gibt zu entscheiden, ob ein Startknoten s und ein Zielknoten t „genügend weit“ voneinander entfernt sind.[4] Dann muss es Zugangsknoten a_s und a_t geben, so dass $\mu(t) = \mu(a_s) + \mu(a_s, a_t) + \mu(a_t, t)$ gilt. Alle hier vorkommenden Abstände sind vorberechnet und stehen in einer Tabelle bereit.

[4] Eventuell ist hierfür eine weitere Vorberechnung nötig.

Es gibt nur etwa 10 Kandidaten für a_s bzw. für a_t, d. h., es werden (nur) etwa 100 Zugriffe auf die Distanztabelle benötigt. Dies kann mehr als eine Million Mal schneller sein als der Algorithmus von Dijkstra. Lokale Anfragen kann man mit einer anderen Technik beantworten, die ausnützen, dass Start und Ziel nahe beieinander liegen. Lokale Anfragen können eventuell auch mit zusätzlichen vorberechneten Tabellen abgedeckt werden. In Abb. 10.12 ist ein Beispiel angegeben (aus [17]).

10.9 Implementierungsaspekte

Kürzeste-Wege-Algorithmen arbeiten über der Menge der erweiterten reellen Zahlen $\mathbb{R} \cup \{+\infty, -\infty\}$. Der Wert $-\infty$ kann ignoriert werden, da er nur benötigt wird, wenn es negative Kreise gibt, und auch dann nur für die Ausgabe, s. Abschnitt 10.6. Auch der Wert $+\infty$ ist nicht wirklich notwendig, da $d[v] = +\infty$ genau dann gilt, wenn $\mathit{parent}(v) = \perp$ gilt. Das heißt: Wenn $\mathit{parent}(v) = \perp$, kann man den in $d[v]$ gespeicherten Zahlenwert ignorieren und so agieren, wie es im Fall $d[v] = +\infty$ vorgesehen ist.

Eine verfeinerte Implementierung des Algorithmus von Bellman und Ford [203, 143] führt eine explizite Approximation T des Baums der kürzesten Wege mit. Dieser Baum enthält nur Knoten und Kanten, die von s aus entlang von Kanten der Form $(\mathit{parent}[v], v)$ erreicht werden können (aber nicht unbedingt alle diese Knoten und Kanten). Bei den Relaxierungen in einer Runde werden die Kanten nach Startknoten gruppiert – es werden also, wie im Algorithmus von Dijkstra, *Knoten bearbeitet*. Knoten, die in der aktuellen Runde noch bearbeitet werden müssen, stehen in einer Menge Q. Wird nun bei der Relaxierung einer Kante $e = (u, v)$ der Wert $d[v]$ verringert, dann werden alle Nachfahren v in T im weiteren Verlauf einen neuen, kleineren d-Wert erhalten. Es ist also überflüssig, diese Knoten mit ihren aktuellen d-Werten zu bearbeiten, und man kann sie aus Q und auch aus T entfernen. Zudem können negative Kreise entdeckt werden, indem man prüft, ob v in T Vorfahr von u ist.

10.9.1 C++

LEDA [130] hat spezielle Klassen *node_pq* für Prioritätswarteschlangen, deren Einträge Graphknoten sind. Sowohl LEDA als auch die Boost-Graphen-Bibliothek [29] enthalten Implementierungen für den Algorithmus von Dijkstra und den Algorithmus von Bellman und Ford. Es gibt einen Graphen-Iterator auf der Basis des Dijkstra-Algorithmus, der es erlaubt, den Suchprozess in flexiblerer Weise zu steuern. Beispielsweise kann man die Suche nur so lange laufen lassen, bis eine vorgegebene Menge von Zielknoten erreicht worden ist. LEDA stellt auch eine Funktion zur Verfügung, die die Korrektheit von Distanzfunktionen verifiziert (s. Aufgabe 10.8).

10.9.2 Java

JDSL [87] stellt eine Implementierung des Algorithmus von Dijkstra für ganzzahlige Kantenkosten bereit. Diese Implementierung erlaubt auch eine detaillierte Steuerung des Suchvorgangs, ähnlich wie die Graph-Iteratoren von LEDA und Boost.

10.10 Historische Anmerkungen und weitere Ergebnisse

Dijkstra [62], Bellman [20] und Ford [69] fanden ihre Algorithmen in den 1950er Jahren. Die ursprüngliche Version des Algorithmus von Dijkstra hatte eine Rechenzeit von $O(m+n^2)$, und es gibt eine lange Geschichte von Verbesserungen. Die meisten davon beruhen auf besseren Datenstrukturen für Prioritätswarteschlangen. Wir habe Binärheaps [222], Fibonacci-Heaps [74], Behälterschlangen [58] und Radix-Heaps [9] diskutiert. Experimentelle Vergleiche lassen sich in [45, 143] finden. Für ganzzahlige Schlüssel sind Radix-Heaps nicht das Ende der Geschichte. Das beste theoretische Resultat ist Zeit $O(m+n\log\log n)$ [209]. Interessanterweise kann man für *ungerichtete* Graphen lineare Zeit erreichen [206]. Der entsprechende Algorithmus bearbeitet Knoten immer noch nacheinander, aber in einer anderen Reihenfolge als der Algorithmus von Dijkstra.

Meyer [151] gab den ersten Kürzeste-Wege-Algorithmus mit linearer mittlerer Rechenzeit an. Der oben beschriebene Algorithmus ALD stammt von Goldberg [83]. Für Graphen mit beschränktem Grad ist der Δ-Stepping-Algorithmus aus [152] sogar noch einfacher. Er benutzt Behälterschlangen und liefert auch einen guten parallelen Algorithmus für Graphen mit beschränktem Grad und kleinem Durchmesser.

Die Ganzzahligkeit von Kantenkosten ist auch hilfreich, wenn negative Kantenkosten erlaubt sind. Wenn alle Kantenkosten ganze Zahlen größer als $-N$ sind, erreicht ein *Skalierungsalgorithmus* Rechenzeit $O(m\sqrt{n}\log N)$ [82].

In Abschnitt 10.8 haben wir einige Beschleunigungstechniken für die Routenplanung skizziert. Hierfür gibt es noch viele andere Techniken. Insbesondere haben wir fortgeschrittene Versionen der zielgerichteten Suche nicht wirklich gewürdigt, ebenso wenig Kombinationen verschiedener Techniken, und andere mehr. Einen recht aktuellen Überblick geben die Arbeiten [181, 188]. Theoretische Rechenzeitgarantien unterhalb derer für den Algorithmus von Dijkstra sind schwieriger zu erreichen. Positive Ergebnisse gibt es für spezielle Graphfamilien wie planare Graphen, oder wenn es ausreicht, nur näherungsweise kürzeste Wege zu berechnen [66, 210, 208].

Es gibt Verallgemeinerungen des Kürzeste-Wege-Problems, bei denen mehrere Kostenfunktionen gleichzeitig betrachtet werden. Möglicherweise möchte man die schnellste Route von einem Startpunkt zu einem Zielpunkt finden, aber unter der Einschränkung, dass kein Nachtanken erforderlich ist. Oder ein Spediteur möchte eine schnellste Route unter der Maßgabe finden, dass ein bestimmter Höchstbetrag für die Autobahnmaut nicht überschritten wird. Solche Kürzeste-Wege-Probleme mit Zusatzeinschränkungen werden in [96, 147] betrachtet.

Kürzeste Wege lassen sich auch in geometrisch beschriebenen Situationen berechnen. Insbesondere gibt es eine interessante Verbindung zur Optik. Verschiedene Materialien haben verschiedene Brechungsindizes, die zur Lichtgeschwindigkeit im jeweiligen Material proportional sind. Die Gesetze der Optik diktieren, dass Lichtstrahlen, die durch verschiedene Materialien gehen, immer entlang von „schnellsten“ Wegen laufen, also solchen, die die Gesamtrechenzeit minimieren.

Aufgabe 10.21. Eine *geordnete Halbgruppe* ist eine Menge S mit einer assoziativen und kommutativen Operation $\oplus$, einem neutralen Element e und einer linearen Ordnung $\leq$, wobei aus $x \leq y$ die Beziehung $x \oplus z \leq y \oplus z$ folgt, für alle x, y und z in

S. (Ein Beispiel für eine solche Halbgruppe ist $S = \mathbb{R}_{\geq 0}$ mit $e = 0$ und $\max\{x, y\}$ als Operation $x \oplus y$.) Welche der Algorithmen in diesem Kapitel funktionieren auch dann, wenn die Kantengewichte aus einer geordneten Halbgruppe kommen? Welche funktionieren unter der zusätzlichen Annahme, dass für alle x in S die Relation $e \leq x$ gilt? (Anmerkung: Ein kürzester Weg bzgl. der max-Operation ist ein *Flaschenhals-Kürzester-Weg*, bei dem die Kosten der teuersten Kante auf dem Weg minimal sind.)