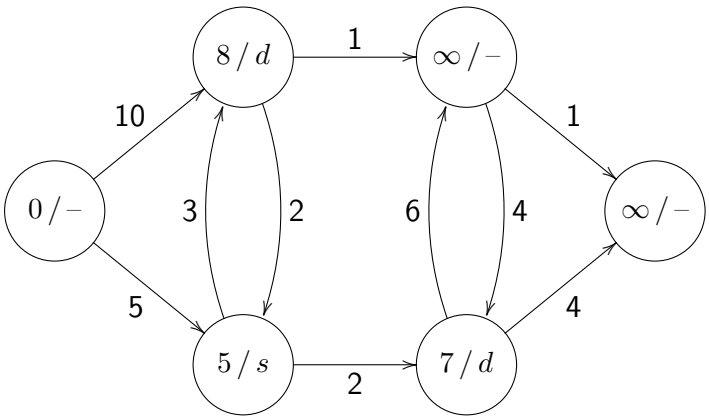
Heiko Körner

Algorithmen

auf

Graphen

Durchmusterungen und kürzeste Wege



Skript zur Vorlesung Algorithmen

IWI Fakultät für Informatik
und Wirtschaftsinformatik

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES
Näher dran.

Vorwort

Das vorliegende Skript soll die Vorlesung ALGORITHMEN unterstützend begleiten, die ich an der Hochschule Karlsruhe laufend anbiete.

Algorithmen sind allgemein Verfahren, mit denen man bestimmte Problemstellungen (wie z.B. das Sortieren von Zahlen) auf einem Computer lösen kann. Die verschiedenen Methoden unterscheiden sich dabei vor allem in ihrer Effizienz, d.h. in dem Zeitaufwand, den man für die Lösung konkreter Probleminstanzen aufbringen muss. So kann man z.B. eine Folge von Zahlen mit den zwei Algorithmen *Bubblesort* oder *Quicksort* sortieren. Das Ergebnis ist dasselbe, aber der *Quicksort*–Algorithmus ist im Allgemeinen viel schneller. Trotzdem wird auf der ganzen Welt unglaublich viel mit Bubblesort sortiert — warum ist das so?

Softwareentwickler stehen in der Praxis häufig unter einem engen Termindruck. Sobald sie auf eine bestimmte Problemstellungen stossen, besorgen sie sich einfach einen passenden und bereits fertig implementierten Algorithmus, z.B. aus dem Internet. Häufig könnte man aber bestimmte Zusatzbedingungen im konkreten Fall gewinnbringend in den Algorithmus einfließen lassen, was aus Zeitgründen jedoch häufig unterbleibt.

Wenn Entwickler „auf die Schnelle“ kein passendes Verfahren finden, programmieren sie sich eben schnell selbst eins — und dabei kommt häufig so etwas wie Bubblesort heraus. Dieses Verfahren ist so einfach, dass „man selbst drauf kommt“, man kann es einfach mit einigen wenigen Programmzeilen implementieren, es ist auch offensichtlich korrekt — aber es ist eben auch sehr ineffizient. Es liegt in der Natur der Sache, dass effiziente Verfahren wie z.B. *Heapsort* komplizierter sind und einem nicht so einfach einfallen. Dafür hat aber das fertige Softwareprodukt später eine ungleich höhere Qualität.

Ziel der Vorlesung ist es, Sie dazu zu ermutigen, in einer solchen Situation *nicht* einfach wie oben beschrieben eine erstbeste Lösung zu implementieren, sondern gezielt sich ein bestimmtes effizientes Verfahren auszusuchen und dieses an die konkreten Gegebenheiten anzupassen. Dazu muss man allerdings wissen, wie man die Korrektheit von Algorith-

men beweisen kann. Nur dann kann man sich nach der Modifikation eines Verfahrens davon überzeugen, dass es immer noch funktioniert. Exakte Korrektheitsbeweise und Laufzeitanalysen werden einen Schwerpunkt dieser Vorlesung bilden.

Aus der Vielzahl von interessanten Fragestellungen wurden für diese Vorlesung zwei zentrale Probleme aus der *Graphentheorie* ausgewählt. Kapitel 1 analysiert effiziente Verfahren, mit denen man die *Konnektivität* von Graphen überprüfen kann. Hierbei geht es darum festzustellen, welche Bereiche eines Graphen von bestimmten anderen Bereichen aus erreichbar sind. Ebenso wird untersucht, wie man die Bestandteile eines Graphen mit unterschiedlichen Methoden durchmustern kann.

Das zweite Kapitel analysiert zwei wichtige Verfahren zur Berechnung von *kürzesten Wegen*. Diese Problemstellung tritt z.B. bei Routenplanern auf, die zwischen zwei Adressen eine beste Verbindung finden müssen.

Mit diesen Algorithmen werden wir nur einen winzigen Bruchteil aller in der Praxis relevanten Problemstellungen abdecken. Mehr ist in einer zweistündigen Vorlesung auch nicht möglich, dazu ist der verfügbare Zeitraum viel zu kurz. Prinzipiell ist dies aber nicht so schlimm, denn der Schwerpunkt der Vorlesung liegt wie gesagt nicht auf den hier konkret behandelten Algorithmen, sondern auf den vorgestellten Beweistechniken — diese sind nämlich auch für viele andere Algorithmen immer wieder die gleichen. Deshalb sollten Sie nach dem erfolgreichen Abschluss in der Lage sein, auch andere Verfahren zu entwerfen bzw. bestehende zu analysieren, so dass Sie das vorgenannte Lernziel erreichen können.

Hinweise zur Vorlesung

Algorithmentheorie ist für viele Studierende ein interessantes, aber auch ein zumindest ungewohntes Fach. Dies liegt an den formalen Beweismethoden, die für die meisten Studierenden neu sind. In vielen anderen Vorlesungen und Laboren sieht und experimentiert man damit, was sich alles mit modernen Computern realisieren lässt. Hier werden wir Algorithmen eher von ihrer theoretischen Seite beleuchten. Man muss also mit logischen Argumenten arbeiten, was manchen anfangs nicht leicht fällt.

Ich empfehle Ihnen, sich dieses Skript anfangs einmal auszudrucken und während der Vorlesung aufgeschlagen vor sich liegen zu haben. Trotzdem sollten Sie die Vorlesungsinhalte im Wesentlichen nur vorne an der Tafel mit verfolgen, d.h. Sie sollten nicht gleichzeitig im Skript mitlesen. Nutzen Sie das Skript in der Vorlesung zunächst nur für eigene Hinweise oder Ergänzungen, die Sie stichpunktartig am Rand notieren. Für Tipps zu Verbesserungsvorschlägen, gefundenen Fehlern usw. bin ich jederzeit sehr dankbar.

Nachmittags oder abends sollten Sie das Gehörte am Skipt dann nochmals nacharbeiten. Dies ist sehr wichtig, denn jede Vorlesung setzt den *gesamten* bereits vermittelten Stoff voraus, und ohne Nacharbeitung wird man sehr schnell „abgehängt“ — keine gute Voraussetzung, um die abschließende Klausur mit einer guten Leistung zu bestehen. Erfahrungsgemäß haben auch die Versuche wenig Erfolg, sich erst wenige Wochen vor

dem Klausurtermin mit dem Lernstoff auseinander zu setzen.

Eine Verfolgung des Skriptes am Bildschirm eines Notebooks hat den Nachteil, dass Sie keine Notizen während der Vorlesung anbringen können. Außerdem verliert man häufig schnell den Anschluss zu dem diskutierten Stoff an der Tafel, weil man bestimmte Details gerade nicht mitbekommen hat. Aus dem gleichen Grund sollten Sie immer möglichst pünktlich zu den Vorlesungen erscheinen.

Die Vorlesung setzt Kenntnisse in der O–Notation sowie in Induktionsbeweisen voraus. Bei fehlenden Kenntnissen wird das Lesen der Anhänge A und B empfohlen, in denen beide Konzepte vorgestellt werden.

Während des Semesters werden Übungsblätter mit mehreren Aufgaben ausgeteilt, die Sie zur Überprüfung Ihrer Kenntnisse verwenden können. Die Lösungen werden zeitgleich auch online gestellt. Die Bearbeitung der Übungsblätter ist keine Pflicht, aber im Hinblick auf die Klausur sehr sinnvoll. Selbstverständlich können Sie auch vor, während oder nach der Vorlesung Fragen stellen.

Die Klausur am Ende des Semesters besteht aus einer Zusammenstellung verschiedener Aufgaben, die in ähnlicher Form auch auf den Übungsblättern zu finden sind. Sobald das Online–Notensystem die Ergebnisse bereit hält, können Sie auf Wunsch Ihre Klausur zu den üblichen Öffnungszeiten im Sekretariat einsehen.

In der letzten Vorlesungswoche wird eine Probeklausur angeboten. Zu allen Klausuren dürfen Sie ein von beiden Seiten beschriftetes DIN A4–Blatt mitnehmen, welches Sie *mit Ihrer eigenen Handschrift* erstellt haben müssen.

Karlsruhe, im Oktober 2009 Heiko Körner

Inhaltsverzeichnis

1 Konnektivität von Graphen

1.1. Graphen und Bäume

Wir wiederholen einige graphentheoretische Grundlagen, die Ihnen zum Teil sicherlich schon bekannt sind.

Graphen sind sehr allgemeine Datenstrukturen, mit deren Hilfe man Abhängigkeiten und Zusammenhänge zwischen einzelnen Komponenten darstellen kann. Sie werden in der Informatik vielfältig eingesetzt, so z.B. zur Beschreibung von Hardwarestrukturen, Rechnernetzen, Datenflüssen, parallelen Systemen und hierarchischen Strukturen.

1.1.1 Definition Ein *ungerichteter Graph* $G = (V, E)$ besteht aus einer endlichen Menge V von *Knoten* und einer endlichen Menge E von 2–elementigen Teilmengen von V, den *Kanten*. Ist $\{v, w\} \in E$, so sagen wir: v und w sind durch eine Kante *verbunden*, *adjazent* oder *benachbart*. Wir schreiben dann:

$$v \text{ — } w \ .$$

Im Zusammenhang mit Graphen bezeichnet üblicherweise $n := |V|$ die Anzahl von Knoten und $m := |E|$ die Anzahl von Kanten.

Eine Folge $(v_0, \ldots, v_k)$ in V ist ein *Weg* der *Länge* k in G, wenn für alle $0 \leq i < k$ gilt: $\{v_i, v_{i+1}\} \in E$, anschaulich:

$$v_0 \text{ — } v_1 \text{ — } \ldots \text{ — } v_k \ .$$

Bei der Länge zählt man also die Anzahl der durchlaufenen Kanten, nicht die Anzahl der durchlaufenen Knoten.

Der Weg zwischen v_0 und v_k ist *geschlossen*, wenn v_0 mit v_k identisch ist. Ein geschlossener Weg heißt auch *Kreis* oder *Zyklus*. Wir vereinbaren, dass ein Weg (abgesehen von dem Start– und Endknoten bei Kreisen) keine doppelten Knoten enthalten darf. Weiterhin sollen Kreise mindestens drei Knoten beinhalten, damit triviale Kreise der Form v — w — v ausgeschlossen sind. Ein Graph ohne Zyklen heißt *azyklisch* oder *kreisfrei*.

1.1.2 Definition Ein ungerichteter Graph G ist *zusammenhängend*, wenn es zwischen je zwei verschiedenen Knoten in G mindestens einen Weg gibt.

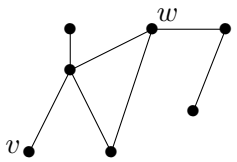
1.1.3 Beispiel Betrachten Sie die beiden folgenden Graphen:

Der linke Graph ist zusammenhängend und besitzt einen Kreis der Länge 3. Der rechte Graph ist nicht zusammenhängend und kreisfrei.

Offenbar kann man bei einem zusammenhängenden Graph mit einem Kreis eine beliebige Kreiskante entfernen; der resultierende Graph ist dann immer noch zusammenhängend.

1.1.4 Definition Der *Grad* eines Knotens v ist die Anzahl seiner *Nachbarn* und wird mit $deg(v)$ bezeichnet.

1.1.5 Beispiel Im obigen linken Graphen haben wir zwei Knoten markiert:



Knoten v hat den Grad 1 und w den Grad 3.

Gerichtete (z.B. zeitliche) Abhängigkeiten kann man mit ungerichteten Graphen nicht beschreiben. Hierfür verwendet man *gerichtete Graphen*.

1.1.6 Definition Bei einem *gerichteten Graph* (*Digraph*, engl.: „*directed graph*“) sind die Kanten nicht (ungeordnete) 2–elementige Teilmengen von V, sondern geordnete Paare aus $V \times V$. Die Kanten (v, w) und (w, v) sind also verschieden, falls $v \neq w$. Eine gerichtete Kante (v, w) veranschaulicht man durch einen Pfeil

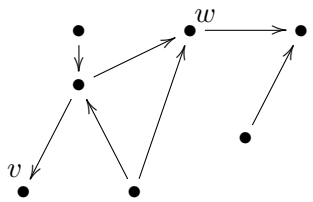
$$v \rightarrow w \ .$$

Bei gerichteten Graphen unterscheidet man zwischen dem *Eingangsgrad* (= Anzahl der eingehenden Kanten) und dem *Ausgangsgrad* (= Anzahl der ausgehenden Kanten) eines Knotens. Für $v \in V$ wird der Ein– und Ausgangsgrad mit $indeg(v)$ bzw. $outdeg(v)$ bezeichnet. Wege in gerichteten Graphen heißen auch *Pfade*.

Bei Kreisen in gerichteten Graphen beträgt die Mindestanzahl an beteiligten Knoten nur zwei, da eine Kante von einem Knoten v zu einem anderen Knoten w nicht automatisch

(wie bei ungerichteten Graphen) eine Kante von w nach v impliziert. Bei Kreisen mit mindestens zwei Knoten müssen in gerichteten Graphen also auch immer mindestens zwei verschiedene Kanten beteiligt sein.

1.1.7 Beispiel Der Graph besitzt nun gerichtete Kanten:



Es gilt $indeg(v) = 1$, $outdeg(v) = 0$, $indeg(w) = 2$, und $outdeg(w) = 1$.

Nachdem wir nun die grundlegenden Begriffe in Zusammenhang mit Graphen eingeführt haben, stellt sich die Frage, wie man Graphen im Rechner effizient darstellen kann. Wir betrachten dazu für jeden Knoten v die Menge ℓ_v aller direkt von v aus erreichbaren Knoten. Es gilt also

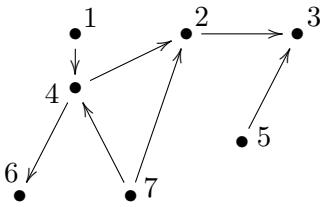
$$\ell_v = \{w \,|\, (v, w) \in E\}$$

im gerichteten bzw.

$$\ell_v = \{w \,|\, \{v, w\} \in E\}$$

im ungerichteten Fall. Normalerweise erfolgt die Verwaltung aller Mengen ℓ_v durch einfach verkettete Listen, den sogenannten *Adjazenzlisten*.

1.1.8 Beispiel Wir greifen noch einmal auf unser Beispiel zurück und haben nun alle Knoten durchnumeriert:



Dann gilt $\ell_1 = \{4\}$, $\ell_2 = \{3\}$, $\ell_3 = \emptyset$, $\ell_4 = \{2, 6\}$, $\ell_5 = \{3\}$, $\ell_6 = \emptyset$ und $\ell_7 = \{2, 4\}$.

Der Vorteil von Adjazenzlisten liegt in dem geringen Speicherbedarf. Im ungerichteten Fall wird jede Kante $\{v, w\} \in E$ implizit nur zweimal aufgeführt, nämlich in ℓ_v und ℓ_w. Im gerichteten Fall wird $(v, w) \in E$ sogar nur einmal in ℓ_v referenziert. Daher gilt

$$\sum_{v \in V} |\ell_v| = 2|E| \quad \text{bzw.} \quad \sum_{v \in V} |\ell_v| = |E| \ ,$$

d.h. es wird in beiden Fällen nur $O(|V|+|E|)$ Speicher benötigt. (Die O–Notation schätzt Aufwände asymptotisch ab, eine Einführung dazu wird in Anhang A gegeben.) Nachteilig

ist, dass Adjazenzlisten nur sequenziell durchlaufen werden können. Dies reicht für die meisten Anwendungen jedoch aus.

1.1.9 Definition Sei $G = (V, E)$ ein ungerichteter Graph. Eine *Zusammenhangskomponente* von G ist eine Teilmenge $Z \subseteq V$ von Knoten, so dass gilt:

- $\forall v, w \in Z$: es existiert in G ein Weg von v nach w
- $\forall v \notin Z$: $\exists w \in Z$: es existiert in G kein Weg von v nach w

In einer Zusammenhangskomponente sind also alle Knoten paarweise über Wege miteinander verbunden, und die Komponente ist bzgl. dieser Eigenschaft knotenmaximal gewählt: jeder weitere Knoten hat keine Verbindung zur Komponente.

1.1.10 Beispiel Nachfolgend sind erneut die Graphen aus Beispiel 1.1.3 aufgeführt, wobei im linken Graphen zwei Kanten entfernt wurden:

Der linke Graph hat nun drei Zusammenhangskomponenten, der rechte zwei.

Offenbar gilt stets:

$$G \text{ ist zusammenhängend} \iff G \text{ besitzt genau eine Zusammenhangskomponente} .$$

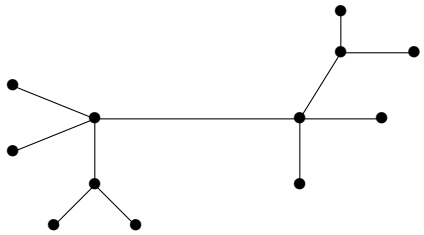
Die Zusammenhangskomponenten von kreisfreien Graphen besitzen eine interessante Eigenschaft:

1.1.11 Satz Sei $G = (V, E)$ ein kreisfreier ungerichteter Graph mit n Knoten, m Kanten und p Zusammenhangskomponenten. Dann gilt $n = m + p$.

Beweis: Durch Induktion über m (eine Einführung in Induktionsbeweise kann Anhang B entnommen werden). Falls $m = 0$, so besitzt der Graph keine Kanten, d.h. er besteht aus n isolierten Knoten und deshalb aus n Zusammenhangskomponenten. Also gilt $p = n$ und die Aussage $n = m + p$ ist richtig.

Sei nun die Aussage für m Kanten bereits richtig. Wenn wir eine weitere Kante $\{v, w\}$ in den Graphen einfügen und der Graph dabei trotzdem kreisfrei bleibt, so können v und w nicht in der gleichen Zusammenhangskomponente liegen. Denn hier existiert bereits ein Weg von w nach v, d.h. die zusätzliche Kante würde einen Kreis erzeugen. Also muss die Kante zwei bisher getrennte Zusammenhangskomponenten zu einer größeren neuen verbinden. Die Anzahl der Kanten steigt um eins, die Anzahl der Zusammenhangskomponenten sinkt um eins, d.h. die Summe $(m + 1) + (p - 1) = m + p$ bleibt unverändert gleich n. □

1.1.12 Definition Ein ungerichteter Graph heißt *Baum*, wenn er zusammenhängend und azyklisch ist:



Bäume gehören zu den wichtigsten Datenstrukturen in der Informatik. Sie eignen sich besonders zur Darstellung hierarchisch strukturierter Daten, die in vielen Anwendungen auftreten. Oft bilden Bäume die Grundlage für effiziente algorithmische Lösungen von Problemen.

Die wesentlichen Eigenschaften eines Baumes fasst der folgende Satz zusammen.

1.1.13 Satz Sei $G = (V, E)$ ein ungerichteter Graph mit n Knoten. Dann sind äquivalent:

a) G ist ein Baum.

b) G ist zusammenhängend und hat genau $n - 1$ Kanten.

c) G ist zusammenhängend, aber nach der Entnahme einer beliebigen Kante gilt dies nicht mehr.

d) Durch das Hinzufügen einer beliebigen Kante entsteht genau ein Kreis.

e) Es gibt für alle $v, w \in V$ genau einen Weg von v nach w.

f) G ist kreisfrei und hat genau $n - 1$ Kanten.

Beweis: Wir zeigen die zyklische Beweiskette a) $\Rightarrow$ e) $\Rightarrow$ c) $\Rightarrow$ d) $\Rightarrow$ f) $\Rightarrow$ b) $\Rightarrow$ a).

„a) $\Rightarrow$ e)“: G ist zusammenhängend, also gibt es für alle Knoten mindestens einen verbindenden Weg. Angenommen, es gibt für zwei Knoten v und w zwei verschiedene Wege von v nach w:

$$v = v_1 — v_2 — \ldots — v_k = w$$

$$v = w_1 — w_2 — \ldots — w_\ell = w$$

Dann können wir prinzipiell beide Wege an den Endpunkten zusammenfügen und erhalten so einen Kreis. Die Wege können aber gemeinsame Anfangs– und Endstücke aufweisen, die zuerst entfernt werden müssen. Wir konstruieren den Kreis daher formal exakt wie folgt:

Sei i minimal gewählt, so dass $v_i \neq w_i$ gilt. Der Index i existiert, denn sonst wären beide Wege gleich. Sei $j \geq i$ minimal gewählt, so dass $v_j \in \{w_i, w_{i+1}, \ldots, w_\ell\}$ gilt. Der Index j existiert, da wegen $w = v_k = w_\ell$ der Knoten v_k in $\{w_i, w_{i+1}, \ldots, w_\ell\}$ enthalten ist. Sei

etwa $v_j = w_r$, $r \in \{i, \ldots, \ell\}$. Gemäß der Wahl von i und j sind dann folgende Aussagen gültig:

- Keiner der Knoten $v_i, v_{i+1}, \ldots, v_{j-1}$ ist in $\{w_i, w_{i+1}, \ldots, w_\ell\}$ enthalten. Insbesondere sind die beiden Mengen $\{v_i, v_{i+1}, \ldots, v_{j-1}\}$ und $\{w_i, w_{i+1}, \ldots, w_r\}$ disjunkt. (Beachten Sie, dass die erste Menge im Fall $i = j$ aber auch leer sein kann.)
- Es kann nicht gleichzeitig $j = i$ und $r = i$ gelten. Denn nach der Wahl von i wäre dann $v_i \neq w_i$, und nach der Wahl von j und r wäre $v_i = w_i$. Es gilt also $(j - i) + (r - i) \geq 1$, und die Menge

  $$\{v_i, v_{i+1}, \ldots, v_{j-1}\} \cup \{w_i, w_{i+1}, \ldots, w_r\}$$

  enthält mindestens zwei Knoten.

Dann aber bildet

$$v_{i-1} — v_i — v_{i+1} — \ \ldots\ — v_j = w_r — w_{r-1} — w_{r-2} — \ \ldots\ — w_i — w_{i-1} = v_{i-1}$$

einen Zyklus der Länge $(j - (i-1)) + (r - (i-1)) = (j-i) + (r-i) + 2 \geq 3$ in G, was der Kreisfreiheit von G widerspricht.

„e) ⇒ c)“: Wenn es zwischen je zwei Knoten einen verbindenden Weg gibt, so ist G zusammenhängend. Sei nun $\{v, w\}$ eine beliebige Kante in G. Dies ist nach Voraussetzung gleichzeitig der einzige Weg von v nach w. Nach der Entnahme dieser Kante gibt es deshalb keinen Weg mehr, d.h. der Graph ist nicht mehr zusammenhängend.

„c) ⇒ d)“: Sei $\{v, w\}$ eine neue Kante. Es gab schon zuvor einen Weg von w nach v, d.h. nun existiert ein Kreis. Vorher kann kein Kreis existiert haben, denn ansonsten hätte man eine Kreiskante entnehmen können, und der Graph wäre immer noch zusammenhängend gewesen. Also entsteht durch das Hinzufügen einer Kante genau ein Kreis.

„d) ⇒ f)“: G ist ohne Zusatzkanten azyklisch. Es kann nur eine Zusammenhangskomponente geben, denn ansonsten könnte man zwei Komponenten mit einer zusätzlichen Kante verbinden, ohne dass ein Kreis entstehen würde. Nach Satz 1.1.11 besitzt G demnach $n - 1$ Kanten.

„f) ⇒ b)“: G besitzt $n-1$ Kanten und ist kreisfrei. Nach Satz 1.1.11 besitzt G demnach genau eine Zusammenhangskomponente, d.h. G ist zusammenhängend.

„b) ⇒ a)“: Angenommen, G enthält Kreise. Wir löschen nach und nach Kanten aus G, so dass zwar alle Kreise verschwinden, aber G zusammenhängend bleibt. Angenommen, wir haben k Kanten gelöscht. Da G immer noch zusammenhängend ist, besitzt G nach Satz 1.1.11 dann genau $n-1$ Kanten — genauso viel wie vorher. Also muss $k = 0$ gelten, d.h. es wurden gar keine Kanten entfernt und G war bereits vorher kreisfrei. □

Wir können nun die ersten elementaren Problemstellungen besprechen sowie die dazu passenden Lösungsalgorithmen diskutieren. Im folgenden sei immer ein Graph (gerichtet oder ungerichtet) $G = (V, E)$ mit n Knoten und m Kanten gegeben.

1.2. Durchmusterungsmethoden

Wir befassen uns als erstes mit der Frage, welche Knoten von einem gegebenen *Startknoten* $s \in V$ aus *erreichbar* sind, d.h. für welche Knoten $v \in V$ ein Weg von s nach v existiert. Alle erreichbaren Knoten sollen markiert werden.

Zwei häufig verwendete Algorithmen für dieses Problem sind die sogenannte *Tiefensuche* (engl. „*Depth–First–Search*“ (*DFS*)) und die sogenannte *Breitensuche* (engl. „*Breadth–First–Search*“ (*BFS*)).

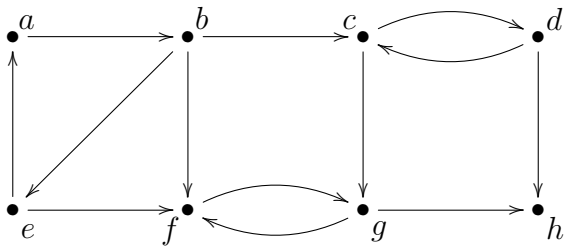
Beide beruhen auf der einfachen Tatsache, dass für zwei adjazente Knoten v und w die Erreichbarkeit von v die Erreichbarkeit von w impliziert. Entsprechend sind alle Nachbarn von w dann ebenfalls erreichbar, usw. Die folgende (abstrakte) Methode verwaltet hierfür eine Menge Q, die stets alle diejenigen Knoten enthält, die bereits als erreichbar erkannt (d.h. markiert) wurden, deren Nachbarknoten aber noch nicht vollständig untersucht worden sind.

```
ALGORITHMUS GRAPHDURCHMUSTERUNG

1    Q := {s};
2    Markiere s;
3    N[s] := Menge ℓ_s aller Nachfolgerknoten von s;
4    while Q ≠ ∅ do {
5       Wähle v ∈ Q;
6       if N[v] ≠ ∅ then {
7          Wähle v' ∈ N[v];
8          N[v] := N[v] \ {v'};
9          if (v' ist noch nicht markiert) then {
10            Q := Q ∪ {v'};
11            Markiere v';
12            N[v'] := Menge ℓ_v' aller Nachfolgerknoten von v';
13         }
14      } else Q := Q \ {v};
15   }
```

Wir betrachten zunächst ein Beispiel für die Ausführung von diesem Algorithmus.

1.2.1 Beispiel Wir wenden den Algorithmus GRAPHDURCHMUSTERUNG auf den Graph auf der nächsten Seite oben an. Als Startknoten sei d vorgesehen.

Zu Beginn markiert der Algorithmus in den ersten drei Zeilen den Startknoten d und



initialisiert die beiden Mengen Q sowie $N[d]$:

$$Q = \{d\} \qquad N[d] = \{c, h\}$$

Q ist also nicht leer, d.h. der Rumpf der `while`–Schleife wird ausgeführt. In Zeile 5 wird d „ausgewählt“ — eine Wahlmöglichkeit besteht in Wirklichkeit natürlich nicht — und die Bedingung in Zeile 6 überprüft. Sie ist wegen $N[d] = \{c, h\} \neq \emptyset$ erfüllt, und in Zeile 7 wird z.B. c als Knoten ausgewählt und anschließend in Zeile 8 aus $N[d]$ entfernt. Da c noch unmarkiert ist, wird in den Zeilen 10 und 11 diese Markierung nun vorgenommen und c zu Q hinzugefügt. Zudem wird die Menge der Nachbarknoten von c in Zeile 12 mit $N[c] := \{d, g\}$ initialisiert. Jetzt gilt also

$$Q = \{c, d\} \qquad N[c] = \{d, g\} \qquad N[d] = \{h\}$$

Beim nächsten Durchlauf der `while`–Schleife kann nun c oder d in Zeile 5 gewählt werden. Wir nehmen für dieses Beispiel an, dass es sich um den Knoten c handelt. Dann erfolgt analog zum ersten Durchlauf der Schleife in Zeile 7 die Auswahl und Entnahme eines Knotens aus $N[c]$ (z.B. g), die Hinzunahme von g zu Q in Zeile 10, die Markierung von g in Zeile 11, sowie die Initialisierung von $N[g]$ mit $\{f, h\}$ in Zeile 12. Wir erhalten daraus die folgende Konfiguration:

$$Q = \{c, d, g\} \qquad N[c] = \{d\} \qquad N[d] = \{h\} \qquad N[g] = \{f, h\}$$

Im nächsten Durchlauf könnte z.B. erneut d in Zeile 5 benutzt werden. Da $N[d]$ nur noch h enthält, wird zwingend h in Zeile 7 ausgewählt, aus $N[d]$ entfernt und — da h noch unmarkiert ist — jetzt markiert sowie zu Q hinzugefügt. Beachten Sie, dass $N[h] = \emptyset$ ist, weil h keine Nachfolger besitzt. Also gilt jetzt

$$Q = \{c, d, g, h\} \qquad N[c] = \{d\} \qquad N[d] = \emptyset \qquad N[g] = \{f, h\} \qquad N[h] = \emptyset$$

Möglicherweise wird im nächsten Durchgang nochmals d in Zeile 5 betrachtet. Dann ist die Bedingung in Zeile 6 zum ersten Mal nicht erfüllt, denn $N[d]$ ist mittlerweile leer. Folglich wird d in Zeile 14 aus Q entfernt. Wir brauchen zudem $N[d]$ nicht mehr in unserer Konfiguration aufzuführen, da markierte Knoten wie d nie mehr erneut nach Q gelangen können und somit keine Rolle mehr spielen. Deshalb gilt für die erreichte Konfiguration jetzt

$$Q = \{c, g, h\} \qquad N[c] = \{d\} \qquad N[g] = \{f, h\} \qquad N[h] = \emptyset$$

Wir gehen nun davon aus, dass in Zeile 5 als nächster Knoten h ausgewählt wird. Dieser Knoten erleidet wegen $N[h] = \emptyset$ das gleiche „Schicksal“ wie d, d.h. er wird aus Q entfernt:

$$Q = \{c, g\} \qquad N[c] = \{d\} \qquad N[g] = \{f, h\}$$

Der Knoten g wurde bislang in Zeile 5 noch nicht betrachtet — angenommen, dies ist jetzt der Fall. Wenn in Zeile 7 dann h aus $N[g]$ ausgewählt und entfernt wird, so wird in Zeile 9 festgestellt, dass dieser Knoten bereits markiert ist, d.h. es werden keine weiteren Aktionen vorgenommen. Wir haben dann als Konfiguration diesen Zustand erreicht:

$$Q = \{c, g\} \qquad N[c] = \{d\} \qquad N[g] = \{f\}$$

Genau das gleiche passiert, wenn anschließend c in Zeile 5 und erzwungenermaßen d in Zeile 7 ausgewählt wird, denn d ist bereits markiert:

$$Q = \{c, g\} \qquad N[c] = \emptyset \qquad N[g] = \{f\}$$

Wird c nochmals ausgewählt, führt dies wegen $N[c] = \emptyset$ zu einem Verlust von c aus Q (Zeilen 6 und 14):

$$Q = \{g\} \qquad N[g] = \{f\}$$

Jetzt ist nur noch g in Q enthalten. Auch die Wahl von f in Zeile 7 ist eindeutig. Dieser Knoten wurde allerdings noch nie betrachtet und wird deshalb neu in Q aufgenommen. Seine Nachbarknotenmenge $N[f]$ enthält nur g:

$$Q = \{f, g\} \qquad N[f] = \{g\} \qquad N[g] = \emptyset$$

Falls der neue Knoten f gleich anschließend in Zeile 5 ausgewählt wird, so wird nach der (eindeutigen) Wahl von g in Zeile 7 und der Entfernung von g aus $N[f]$ entdeckt, dass g bereits markiert ist (Zeile 9). Deshalb erfolgt keine Neuaufnahme von g nach Q. Wir erhalten:

$$Q = \{f, g\} \qquad N[f] = \emptyset \qquad N[g] = \emptyset$$

Beide in Q verbliebenen Knoten f und g besitzen jetzt nur noch leere Nachbarknotenmengen. Deshalb werden beide Knoten in den folgenden beiden Durchläufen der `while`–Schleife entfernt (siehe Zeilen 6 und 14). Zuerst trifft es z.B. g, also

$$Q = \{f\} \qquad N[f] = \emptyset$$

und danach f:

$$Q = \emptyset$$

Jetzt ist Q leer, und das Verfahren terminiert. Insgesamt wurden die Knoten c, d, f, g, und h markiert. Es ist leicht zu sehen, dass nur genau diese Knoten von d aus erreichbar sind. Dies ist kein Zufall, wie der nachfolgende Korrektheitsbeweis zeigt.

1.2.2 Satz Nach Terminierung des Algorithmus GRAPHDURCHMUSTERUNG sind genau alle von s aus erreichbaren Knoten markiert.

Beweis: Nehmen wir an, ein von s aus erreichbarer Knoten v ist nicht markiert worden. Sei $s = v_0 \to v_1 \to \ldots \to v_k = v$ (bzw. $s = v_0 — v_1 — \ldots — v_k = v$ im ungerichteten Fall) ein Weg von s nach v. Es sei v_i der erste nicht markierte Knoten auf diesem Weg. v_i existiert, da v_k nicht markiert worden ist. Ferner gilt $i \neq 0$, da s in Zeile 2 markiert wird. Der vor v_i liegende Knoten v_{i-1} ist markiert und somit in Q aufgenommen worden. Da zum Abschluss des Programms $Q = \emptyset$ gilt, muss v_{i-1} zuvor in Zeile 14 wieder aus Q entfernt worden sein. Dies impliziert $N[v_{i-1}] = \emptyset$ (Zeile 6). Da anfangs v_i in $N[v_{i-1}]$ enthalten war (Zeilen 3 und 12), muss v_i in Zeile 7 ausgewählt und in Zeile 8 aus $N[v_{i-1}]$ entfernt worden sein. Anhand der Zeilen 9–13 ist aber klar, dass v_i nach Durchlauf dieser Zeilen auf jeden Fall markiert ist. Dies ist ein Widerspruch zur Wahl von v_i.

Umgekehrt nehmen wir nun an, dass zumindest ein Knoten markiert wurde, der nicht von s aus erreichbar ist. Sei v' der erste Knoten, der während des Programmablaufs fälschlicherweise in Zeile 11 markiert wird. Aufgrund der Arbeitsweise des Programms (Zeilen 5 und 7) folgt $v' \in N[v]$ für einen zuvor markierten Knoten v. Gemäß der Wahl von v' ist die Markierung von v korrekt, d.h. v ist von s aus erreichbar, somit also auch v'. Dies ist ein Widerspruch zur Wahl dieses Knotens. □

1.2.3 Satz Sei $V' \subseteq V$ die Menge aller von s aus erreichbaren Knoten und (im gerichteten bzw. ungerichteten Fall) $E' := E \cap (V' \times V')$ bzw. $E' := E \cap \{\{v, w\} \mid v, w \in V'\}$ die Menge aller Kanten, die ausschließlich zwischen Knoten aus V' verlaufen. Das obige Verfahren kann dann mit einer Laufzeit von nur $O(|V'| + |E'|)$ implementiert werden.

Beweis: Nur unmarkierte Knoten können in Q aufgenommen werden, dabei werden sie markiert. Da eine Markierung nie mehr entfernt wird, kann jeder Knoten höchstens einmal nach Q gelangen. Wird ein solcher Knoten v in Zeile 5 ausgewählt und nicht direkt in Zeile 14 entfernt, so schrumpft $N[v]$ wegen Zeile 8 um ein Element. Also wird v nach der $(outdeg(v) + 1)$–sten Betrachtung (bzw. $(deg(v) + 1)$–sten Betrachtung im ungerichteten Fall) entfernt.

Die Anzahl der Durchläufe der `while`–Schleife ist deshalb durch

$$\sum_{v \in V'} (outdeg(v) + 1) = |E'| + |V'|$$

im gerichteten bzw. durch

$$\sum_{v \in V'} (deg(v) + 1) = 2|E'| + |V'|$$

im ungerichteten Fall begrenzt. Der Rumpf der `while`–Schleife erfordert dabei pro Durchlauf nur konstant viel Zeit, denn die Entnahme eines Knotens aus $N[v]$ kann ebenfalls in konstanter Zeit ausgeführt werden. Wir verwalten hierfür für jede Adjazenzliste ℓ_v einen Zeiger, der auf den ersten noch nicht abgearbeiteten Knoten von ℓ_v verweist. □

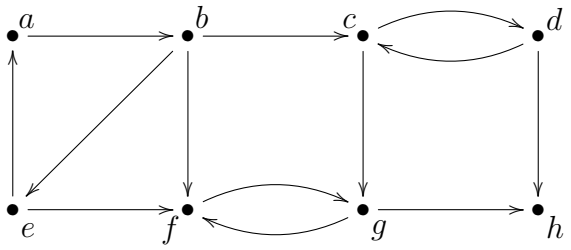
Beachten Sie, dass damit der Durchmusterungsalgorithmus zeitoptimal ist. Denn auch jeder andere Algorithmus muss sich alle erreichbaren Knoten und die dazugehörigen Kanten zumindest einmal ansehen (wenn er es nicht tut, verpasst er womöglich einen erreichbaren Knoten). Allein dafür werden aber bereits $\Omega(|V'| + |E'|)$ viele Schritte benötigt.

Wir haben offen gelassen, wie die Menge Q verwaltet werden soll. Wählen wir als zugrundeliegende Datenstruktur einen *Keller* (engl. „*Stack*“), so erhalten wir *Depth–First–Search* (*DFS*). Wird Q dagegen als *Warteschlange* (engl. „*Queue*“) verwaltet, so erhalten wir *Breadth–First–Search* (*BFS*). Wir analysieren zunächst DFS.

Durch die Verwendung eines Kellers Q bekommt der Durchmusterungsalgorithmus das nachfolgende Aussehen. Wir verwalten zusätzlich eine „Uhr“ u, die jede $Push$– und Pop–Operation eines Knotens v mit Zeitmarken $d[v]$ und $f[v]$ versieht.

```
ALGORITHMUS DFS

1    Push(Q, s);
2    Markiere s;
3    N[s] := Menge ℓ_s aller Nachfolgerknoten von s;
4    u := 1;
5    d[s] := u;
6    while not Stack-empty(Q) do {
7       v := top(Q);
8       if N[v] ≠ ∅ then {
9          Wähle v' ∈ N[v];
10         N[v] := N[v] \ {v'};
11         if (v' ist noch nicht markiert) then {
12            Push(Q, v');
13            Markiere v';
14            N[v'] := Menge ℓ_v' aller Nachfolgerknoten von v';
15            u := u + 1;
16            d[v'] := u;
17         }
18      } else {
19         Pop(Q);
20         u := u + 1;
21         f[v] := u;
22      }
23   }
```

1.2.4 Beispiel Angenommen, DFS wird auf den gleichen Graph wie in Beispiel 1.2.1 angewendet (als Startknoten ist diesmal c vorgesehen):



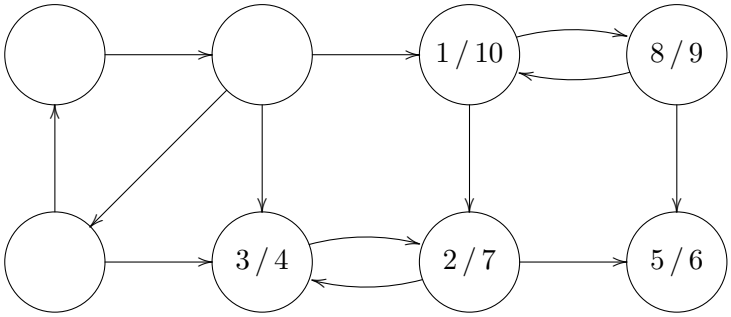
Der Algorithmus markiert in den ersten fünf Zeilen den Startknoten c, legt c als ersten Knoten auf dem Keller Q ab, initialisiert $N[c]$ mit der Nachbarknotenmenge $\{d, g\}$ und vergibt für c die *Push*–Zeit $d[c] = 1$. Da Q den Knoten c enthält und deshalb nicht leer ist, wird der Rumpf der `while`–Schleife zum ersten Mal betreten und der Knoten c in Zeile 7 ausgelesen. $N[c]$ enthält d und g, ist also nicht leer, und es wird einer der beiden Knoten — zum Beispiel g — in Zeile 9 ausgewählt und in Zeile 10 aus $N[c]$ entfernt. Da g noch nicht markiert ist, wird in den Zeilen 11–17 diese Markierung nun vorgenommen und g über c in Q abgelegt. Zudem wird die Menge der Nachbarknoten von g in Zeile 14 mit $N[g] := \{f, h\}$ initialisiert.

Jetzt steht g oben auf dem Keller, und der nächste Durchlauf der `while`–Schleife verläuft analog. Wir nehmen an, dass diesmal f in Zeile 9 ausgewählt wird. Dieser Knoten wird aus $N[g]$ herausgenommen, landet dann als oberster Knoten auf dem Keller, und wird im nächsten Durchlauf der `while`–Schleife behandelt. Seine Nachbarknotenliste $N[f]$ enthält jedoch nur den bereits markierten Knoten g. Dieser wird in Zeile 9 ausgewählt, in Zeile 10 aus $N[f]$ entfernt, und wegen seiner Markierung und der Bedingung in Zeile 11 ignoriert. Beim nächsten Durchlauf der `while`–Schleife steht dann immer noch f oben auf dem Keller, aber da $N[f]$ nun leer ist, werden die Zeilen 19–21 ausgeführt. Der Knoten f wird demnach vom Keller genommen und die *Pop*–Zeit $f[f] = 4$ vergeben (vorher wurde die Uhr bei jeder *Push*–Operation hochgezählt, also insgesamt dreimal).

Jetzt steht erneut g oben auf dem Keller, und $N[g]$ enthält nur noch den Knoten h. Deshalb wird nun h aus $N[g]$ entnommen und auf den Keller gelegt (sein Inhalt ist jetzt c, g, h), aber bereits beim nächsten Durchgang wieder entfernt, da h keine Nachfolger besitzt, $N[h]$ also leer ist. Die Uhr wird dabei erneut zweimal weitergezählt ($d[h] = 5$, $f[h] = 6$). Da nun auch $N[g]$ leer ist, wird auch g vom Keller gelöscht. Die Uhr steht jetzt bei $u = 7$.

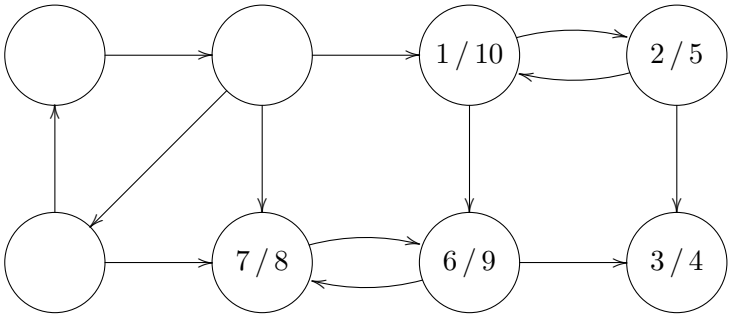
Der Knoten c ist jetzt das einzige Element auf dem Keller Q, und $N[c]$ enthält nur noch den Knoten d. Folglich landet d auf dem Keller, aber ähnlich wie bei f sind die beiden Nachfolger c und h von d bereits markiert, d.h. beide Knoten werden ohne weitere Konsequenzen aus $N[d]$ gelöscht. Anschließend wird wegen $N[d] = \emptyset$ zunächst d und danach wegen $N[c] = \emptyset$ auch c vom Keller entfernt. Weil dann Q leer ist, terminiert der Algorithmus.

Insgesamt gesehen ermittelt DFS hier also die folgenden Suchergebnisse:



Dabei sind für jeden Knoten v die *Push*– und *Pop*–Zeiten als Zahlenpaare $(d[v] \,/\, f[v])$ eingetragen. Leere Knoten wurden nicht gefunden und sind vom Startknoten aus nicht erreichbar.

Da nicht näher bestimmt ist, in welcher Reihenfolge die Nachfolger eines Knotens betrachtet werden, wäre ebenso gut ein anderes Ergebnis möglich:



Wir können nun zwei zentrale Eigenschaften von DFS beweisen.

1.2.5 Lemma Sei während der Programmausführung von DFS $v = top(Q)$. Dann enthält Q einen Weg von s nach v.

Beweis: Angenommen nicht. Sei v' der erste Knoten, der zu einer Verletzung der obigen Behauptung bei der Ausführung von $Push(Q, v')$ führt. Unmittelbar zuvor enthält Q einen Weg von s zum Vorgängerknoten v, und es gilt $v' \in N[v]$, d.h. $v \rightarrow v'$ bzw. $v - v'$. Also enthält Q nach der Ausführung von $Push(Q, v')$ einen gültigen Weg von s nach v'. Dies steht im Widerspruch zur Wahl von v'. □

Die zweite Eigenschaft betrifft die Vergabe der Zeitmarken $d[v]$ und $f[v]$. Natürlich gilt für jeden markierten Knoten v, dass die *Push*–Zeit $d[v]$ kleiner als die *Pop*-Zeit $f[v]$ ist. Es lässt sich aber sogar zeigen:

1.2.6 Satz Es seien u und v zwei von s aus erreichbare Knoten. Dann sind die beiden folgenden Aussagen äquivalent:

a) Zum Zeitpunkt $d[u]$ ist v von u aus auf einem Weg mit ausschließlich unmarkierten Knoten erreichbar (abgesehen von dem soeben markierten Knoten u).

b) Es gilt am Ende $d[u] < d[v] < f[v] < f[u]$.

In der Literatur ist dieser Satz auch als „*White–Path–Theorem*“ bekannt. Der Name ergibt sich aus der Markierung eines gefundenen Knotens, die häufig symbolisch durch einen Anstrich mit schwarzer Farbe vorgenommen wird. Die in der ersten Behauptung erwähnten unmarkierten Knoten sind also noch nicht bemalt worden und gelten deshalb als „weiß“.

Beweis: Wir zeigen zunächst die Hinrichtung „a) ⇒ b)“. Es gilt stets $d[u] < d[v]$, da v zum Zeitpunkt $d[u]$ noch unmarkiert ist. Die restlichen Ungleichungen zeigen wir durch vollständige Induktion über die Länge ℓ des Weges von u nach v.

Falls $\ell = 1$ gilt, so gibt es eine direkte Kante von u nach v. Der Knoten u wird erst dann aus dem Keller entfernt, wenn es keine unmarkierten Nachfolger von u mehr gibt. Insbesondere muss also $d[v] < f[u]$ gelten, d.h. wir haben $d[u] < d[v] < f[u]$. Der Knoten v landet also auf dem Keller, wenn u bereits im Keller steht, und bevor u wieder aus dem Keller entfernt wird. Also muss v auch vor u wieder aus dem Keller entfernt werden, da sonst kein Zugriff auf u besteht. Somit erhalten wir wie gewünscht

$$d[u] < d[v] < f[v] < f[u] \ .$$

Der Weg habe nun die Länge $\ell + 1$, d.h. es gilt

$$u = w_0 \ \rightarrow \ w_1 \ \rightarrow \ \ldots \ \rightarrow \ w_\ell \ \rightarrow \ w_{\ell+1} = v$$

bzw.

$$u = w_0 \ — \ w_1 \ — \ \ldots \ — \ w_\ell \ — \ w_{\ell+1} = v \ .$$

Aus der Induktionshypothese folgt nun $d[u] < d[w_\ell] < f[w_\ell] < f[u]$. Ferner wird der Knoten w_ℓ erst dann aus dem Keller entfernt, wenn es keine unmarkierten Nachfolger mehr gibt. Also gilt $d[w_{\ell+1}] < f[w_\ell]$, wegen $f[w_\ell] < f[u]$ also sogar $d[w_{\ell+1}] < f[u]$. Zudem ist $w_{\ell+1}$ zum Zeitpunkt $d[u]$ noch unmarkiert. Es gilt folglich auch $d[u] < d[w_{\ell+1}]$, insgesamt also

$$d[u] < d[w_{\ell+1}] < f[u] \ .$$

Wegen der Identität $w_{\ell+1} = v$ erhalten wir somit die gleiche Situation $d[u] < d[v] < f[u]$ wie beim Induktionsanfang und können deshalb genauso auf das gewünschte Ergebnis

$$d[u] < d[v] < f[v] < f[u]$$

schließen. Der Induktionsbeweis der Hinrichtung ist damit vollständig.

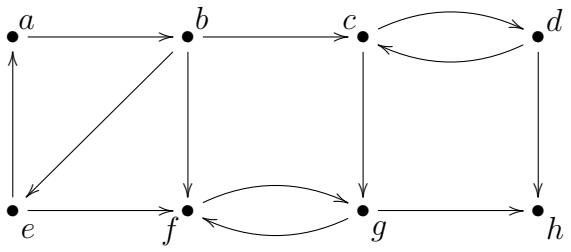
Die Rückrichtung „b) ⇒ a)“ ist leichter zu beweisen. Zum Zeitpunkt $d[v]$ steht v über u auf dem Keller. Lemma 1.2.5 besagt, dass es einen Weg von s über u nach v geben muss, insbesondere also einen Weg von u nach v. Ferner waren alle Knoten über u (d.h. alle anderen Knoten des Weges) zum Zeitpunkt $d[u]$ noch unmarkiert, da nur beim Markierungsvorgang neue Knoten auf den Keller gelegt werden. □

Wir studieren als nächstes das *Breadth–First–Search* (*BFS*) Verfahren. Hier wird, wie bereits erwähnt, die Menge Q als Queue verwaltet. Zusätzlich wird ein Attribut $d[v]$ für

jeden erreichbaren Knoten v festgelegt. Wir werden zeigen, dass für alle erreichbaren Knoten v der Wert $d[v]$ gerade die Länge eines kürzesten Weges von s nach v ist.

```
ALGORITHMUS BFS

 1    Enqueue(Q, s);
 2    Markiere s;
 3    N[s] := Menge ℓ_s aller Nachfolgerknoten von s;
 4    d[s] := 0;
 5    while not Queue-empty(Q) do {
 6       v := head(Q);
 7       if N[v] ≠ ∅ then {
 8          Wähle v' ∈ N[v];
 9          N[v] := N[v] \ {v'};
10          if (v' ist noch nicht markiert) then {
11             Enqueue(Q, v');
12             Markiere v';
13             N[v'] := Menge ℓ_v' aller Nachfolgerknoten von v';
14             d[v'] := d[v] + 1;
15          }
16       } else Dequeue(Q);
17    }
```

1.2.7 Beispiel Wir demonstrieren das Ergebnis von BFS am gleichen Graphen wie in Beispiel 1.2.4 und starten wie dort beim Knoten c:



Wie in Beispiel 1.2.1 geben wir die Konfigurationen an, die jeweils beim Erreichen des Kopfes der `while`–Schleife in Zeile 5 gelten.

$\Downarrow$ Die ersten vier Zeilen werden durchlaufen

$Q = (c)$ $N[c] = \{d, g\}$

$\Downarrow$ In Zeile 6 gilt $head(Q) = c$; wähle z.B. $g \in N[c]$ in Zeile 8

$Q = (c, g)$ $N[c] = \{d\}$ $N[g] = \{f, h\}$

$\Downarrow$ In Zeile 6 gilt $head(Q) = c$; wähle (eindeutig) $d \in N[c]$ in Zeile 8

$Q = (c, g, d)$ $N[c] = \emptyset$ $N[d] = \{c, h\}$ $N[g] = \{f, h\}$

$\Downarrow$ c wird wegen $N[c] = \emptyset$ in Zeile 16 aus Q entfernt

$Q = (g, d)$ $N[d] = \{c, h\}$ $N[g] = \{f, h\}$

$\Downarrow$ In Zeile 6 gilt $head(Q) = g$; wähle z.B. $h \in N[g]$ in Zeile 8

$Q = (g, d, h)$ $N[d] = \{c, h\}$ $N[g] = \{f\}$ $N[h] = \emptyset$

$\Downarrow$ In Zeile 6 gilt $head(Q) = g$; wähle (eindeutig) $f \in N[g]$ in Zeile 8

$Q = (g, d, h, f)$ $N[d] = \{c, h\}$ $N[f] = \{g\}$ $N[g] = \emptyset$ $N[h] = \emptyset$

$\Downarrow$ g wird wegen $N[g] = \emptyset$ in Zeile 16 aus Q entfernt

$Q = (d, h, f)$ $N[d] = \{c, h\}$ $N[f] = \{g\}$ $N[h] = \emptyset$

$\Downarrow$ In Zeile 6 gilt $head(Q) = d$; wähle z.B. markiertes $c \in N[d]$ in Zeile 8

$Q = (d, h, f)$ $N[d] = \{h\}$ $N[f] = \{g\}$ $N[h] = \emptyset$

$\Downarrow$ In Zeile 6 gilt $head(Q) = d$; wähle markiertes $h \in N[d]$ in Zeile 8

$Q = (d, h, f)$ $N[d] = \emptyset$ $N[f] = \{g\}$ $N[h] = \emptyset$

$\Downarrow$ d wird wegen $N[d] = \emptyset$ in Zeile 16 aus Q entfernt

$Q = (h, f)$ $N[f] = \{g\}$ $N[h] = \emptyset$

$\Downarrow$ h wird wegen $N[h] = \emptyset$ in Zeile 16 aus Q entfernt

$Q = (f)$ $N[f] = \{g\}$

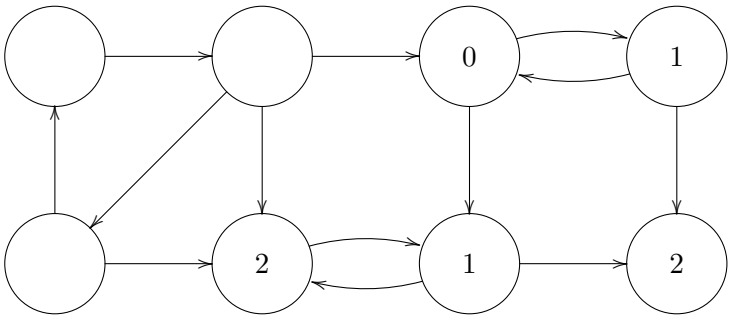
$\Downarrow$ In Zeile 6 gilt $head(Q) = f$; wähle markiertes $g \in N[f]$ in Zeile 8

$Q = (f)$ $N[f] = \emptyset$

$\Downarrow$ f wird wegen $N[f] = \emptyset$ in Zeile 16 aus Q entfernt

$Q = ()$

Wegen der Leere von Q terminiert der Algorithmus an dieser Stelle. Die von BFS ermittelten Distanzen zu s können dem folgenden Graph entnommen werden:



Insbesondere ist im Gegensatz zu DFS das Attributierungsergebnis von BFS immer eindeutig.

Den Beweis der Distanzeigenschaft müssen wir durch zwei Lemmata vorbereiten.

1.2.8 Lemma Sei zu einem beliebigen Zeitpunkt während der Ausführung von BFS $(v_0, v_1, \ldots, v_k)$ der Inhalt der Queue Q. Dann gilt:

- Alle Knoten der Queue sind markiert, und es gilt $d[v_0] \leq d[v_1] \leq \cdots \leq d[v_k]$, d.h. die Knotenattribute liegen in sortierter Reihenfolge vor.
- Für jeden bereits markierten Knoten w gilt $d[w] \leq d[v_0] + 1$.

Beweis: Anfangs enthält Q nur den Startknoten s, und s ist auch der einzige markierte Knoten. Deshalb sind beide Aussagen zu diesem Zeitpunkt sicher richtig.

Q wird nun durch zwei Operationen fortlaufend modifiziert, nämlich durch die Entnahme des aktuellen Kopfknotens v_0 am Anfang oder durch das Anfügen eines neuen Knotens am Ende. In beiden Fällen bleiben die Aussagen jedoch richtig, wie wir jetzt zeigen werden.

Wird der Knoten $v = v_0$ am Anfang entnommen, so ist die restliche Queue sicherlich immer noch sortiert. Für jeden markierten Knoten w galt ferner $d[w] \leq d[v_0] + 1$, also gilt für den neuen Kopfknoten v_1 wegen $d[v_0] \leq d[v_1]$ erst recht $d[w] \leq d[v_1] + 1$. Beide Aussagen sind also immer noch wahr.

Wird ein neuer Knoten v_{k+1} an Q angehängt, so erhält dieser neben seiner Markierung das Attribut $d[v_{k+1}] := d[v_0] + 1$ (siehe Zeilen 6 und 14). Die zweite Aussage ist demnach noch immer richtig. Für den bisherigen Endknoten v_k galt aufgrund der zweiten Aussage zudem $d[v_k] \leq d[v_0] + 1$, d.h. wir haben $d[v_k] \leq d[v_{k+1}]$. Die Attribute liegen also immer noch sortiert vor. Somit bleibt auch die erste Aussage erhalten.

Keine der beiden Aussagen kann also jemals falsch werden. □

1.2.9 Lemma Ist v von s aus erreichbar, so ist die Länge eines kürzesten Weges von s nach v kleiner oder gleich $d[v]$.

Beweis: Wir beweisen induktiv die Gültigkeit der Aussage für alle bereits markierten Knoten. Anfangs ist nur s markiert. Ein kürzester Weg von s nach s hat trivialerweise die Länge 0, und es gilt auch $d[s] = 0$. Die Aussage ist anfangs also richtig.

Wenn ein weiterer Knoten w markiert wird, so geschieht dies aufgrund der Betrachtung eines Vorgängerknotens v von w. Für v ist die Aussage bereits gültig, d.h. es existiert ein Weg mit höchstens $d[v]$ Kanten von s nach v. Wir können diesen Weg um die Kante von v nach w verlängern und schließen daraus, dass ein kürzester Weg von s nach w nicht länger als $d[v] + 1$ sein kann. Dies ist aber gerade der an $d[w]$ in Zeile 14 vergebene Attributwert. □

Die Attribute sind jedoch nicht nur eine Abschätzung für die Weglängen, sondern es gilt sogar die Gleichheit:

1.2.10 Satz Ist v von s aus erreichbar, so enthält $d[v]$ nach der Ausführung von BFS die Länge eines kürzesten Weges von s nach v.

Beweis: Angenommen nicht. Es gibt also eine bestimmte nichtleere Menge von Knoten, die mit einer falschen Weglänge attributiert wurden. Unter diesen sei v ein Knoten mit einer möglichst kurzen (tatsächlichen) Distanz zu s. Bei v kann es sich nicht um den Startknoten s handeln, denn s wird in Zeile 4 sicher mit dem richtigen Attribut versehen. Sei

$$s = w_0 \ \rightarrow \ w_1 \ \rightarrow \ \ldots \ \rightarrow \ w_{k-1} \ \rightarrow \ w_k = v$$

bzw.

$$s = w_0 \ — \ w_1 \ — \ \ldots \ — \ w_{k-1} \ — \ w_k = v$$

ein kürzester Weg von s nach v der Länge k. Der Wert k muss größer als Null sein, denn ansonsten wäre s mit v identisch, was wir gerade ausgeschlossen haben. Es existiert also zumindest der Vorgängerknoten w_{k-1}, und ein kürzester Weg von s nach w_{k-1} umfasst stets $k - 1$ Kanten. Denn zum einen liegt uns ein Weg dieser Länge konkret vor, und zum anderen könnte man einen noch kürzeren Weg um die Kante von w_{k-1} nach $w_k = v$ verlängern und somit auch den Weg nach v verkürzen. Dies aber wäre ein Widerspruch zur obigen Wahl des Weges von s nach v.

Der Knoten v hat die Distanz k zu s, und nach der Wahl von v gibt es keine Knoten, die ebenfalls falsch attributiert und gleichzeitig weniger weit entfernt sind. Folglich muss der Vorgängerknoten w_{k-1} mit der richtigen Weglänge $d[w_{k-1}] = k - 1$ gekennzeichnet sein. Wir betrachten jetzt den Zeitpunkt, wenn w_{k-1} am Kopf der Queue steht und der Nachfolgerknoten $w_k = v$ aus $N[w_{k-1}]$ in Zeile 10 ausgewählt wird.

Falls v zu diesem Zeitpunkt noch nicht markiert ist, so wird die Markierung nun vergeben und v korrekt mit dem Attribut $d[w_{k-1}] + 1 = (k-1) + 1 = k$ versehen, was ein Widerspruch zur Wahl von v ist. Wenn v aber bereits markiert wurde, dann besitzt das zugehörige Attribut $d[v]$ gemäß Lemma 1.2.8 höchstens den Wert

$$d[w_{k-1}] + 1 = (k-1) + 1 = k \ .$$

Gemäß Lemma 1.2.9 ist k zugleich aber auch eine untere Schranke für den Wert $d[v]$. Also kommt nur die korrekte Weglänge k als Wert für das Attribut $d[v]$ in Frage, was ein weiterer Widerspruch ist.

Also kann ein solcher Knoten v gar nicht existieren, d.h. alle Attribute sind korrekt. □

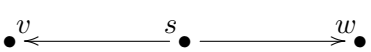
1.3. Starke Zusammenhangskomponenten

Das weltweite Internet kann man sich in Form eines (sehr) großen gerichteten Graphen vorstellen, bei dem jede Webseite einen Knoten repräsentiert. Eine Webseite A besitzt eine Kante zu einer Webseite B, wenn A einen Link auf B enthält. Suchmaschinen benutzen diese Informationen, um bestimmte Seiten auf ihren Ergebnislisten als höherwertiger einzustufen (nämlich dann, wenn viele andere Webseiten auf diese Seiten verweisen).

Thematisch eng verwandte und „gleichberechtigte“ Webseiten kann man gut anhand von „Anhäufungen“ erkennen, d.h. die beteiligten Webseiten lassen sich (evtl. durch Verfolgen mehrerer Links) wechselseitig untereinander von jeder zugehörigen Webseite aus erreichen. Wie kann eine Suchmaschine ein solches Themengebiet erkennen?

Wenn wir einen Link immer als bidirektional (also ungerichtet) ansehen würden, wäre dies mit den bis jetzt schon besprochenen Methoden kein Problem. Sowohl DFS als auch BFS beantworten zwar eigentlich nur die Frage, welche Knoten man von *einem* bestimmten Startknoten aus erreichen kann, in ungerichteten Graphen repräsentiert die gefundene Knotenmenge dann aber sogar schon die zugehörige Zusammenhangskomponente. Denn von jedem ermittelten Knoten aus existiert ein Weg zurück zum Startknoten und weiter zu jedem anderen Knoten in der Komponente. In gerichteten Graphen (und Links sind nun einmal gerichtet) ist dies jedoch nicht richtig, denn zwei Pfade von z.B. s nach v sowie s nach w implizieren nicht unbedingt einen Pfad von v nach w.

1.3.1 Beispiel Betrachten Sie dazu den folgenden Graph:



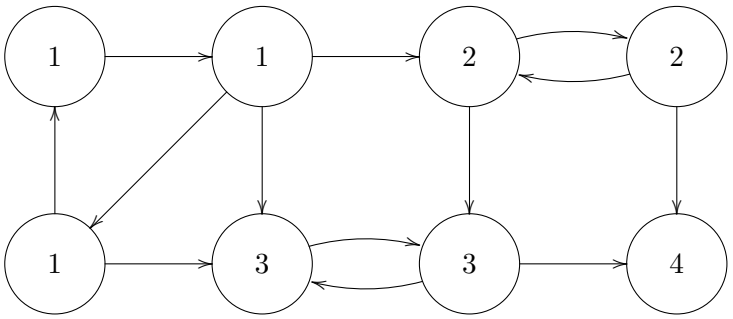
Natürlich sind v und w von s aus erreichbar, dennoch existiert keine Verbindung zwischen diesen beiden Knoten.

1.3.2 Definition Sei $G = (V, E)$ ein gerichteter Graph. Eine *starke Zusammenhangskomponente* ist eine nichtleere Teilmenge $Z \subseteq V$ von Knoten, so dass gilt:

- $\forall v, w \in Z$: es existiert in G ein Pfad von v nach w.
- $\forall v \notin Z$: $\forall w \in Z$: es fehlt in G ein Pfad von v nach w oder umgekehrt ein Pfad von w nach v (oder beides).

In einer starken Zusammenhangskomponente sind also alle Knoten paarweise über Pfade miteinander verbunden, und die Komponente ist bzgl. dieser Eigenschaft knotenmaximal gewählt: für jeden weiteren Knoten fehlt mindestens in der einen oder anderen Richtung eine Verbindung zu der Komponente.

1.3.3 Beispiel Betrachten Sie unseren Graphen aus Beispiel 1.2.1. Der Graph besitzt vier starke Zusammenhangskomponenten. Alle Knoten einer Komponente sind mit der gleichen Zahl beschriftet.



Wie kann man einen gerichteten Graphen in seine starken Zusammenhangskomponenten zerlegen? Ein möglicher Weg wäre, für jeden Knoten $v \in V$ den Algorithmus DFS oder BFS mit v als Startknoten auszuführen und so die Mengen

$$Reach[v] := \{w \mid w \text{ ist von } v \text{ aus erreichbar}\}$$

zu bestimmen. Anschließend kann man testen, zwischen welchen Knoten v und w in beiden Richtungen Wege existieren, indem man die Bedingungen $v \in Reach[w]$ und $w \in Reach[v]$ überprüft. Dies führt zu dem Algorithmus auf der folgenden Seite, der in Zeile 10 nacheinander alle starken Zusammenhangskomponenten ausgibt. Die Menge V' verwaltet dabei die Menge aller Knoten, die bislang noch keiner Komponente angehören.

1.3.4 Satz Der Algorithmus Gegenseitige Erreichbarkeit arbeitet korrekt.

Beweis: Jeder Knoten ist nur in genau einer der in Zeile 10 gemeldeten Komponenten enthalten, denn die Knotenmenge einer ausgegebenen Komponente wird danach in Zeile 11 aus der Menge V' entfernt, aus der alle übrigen Komponenten generiert werden. Sei jetzt C eine beliebige erzeugte Komponente. Für zwei Knoten $w, w' \in C$ existieren wegen den Zeilen 5 und 6 Pfade von v nach w und w' sowie Pfade von w und w' nach v. Hieraus kann man leicht Pfade von w über v nach w' sowie von w' über v nach w generieren. Also

```
Algorithmus Gegenseitige Erreichbarkeit

1    V' := V;
2    while V' ≠ ∅ do {
3       Wähle v ∈ V';
4       C := ∅;
5       for w ∈ Reach[v] do {
6          if v ∈ Reach[w] then {
7             C := C ∪ {w};
8          }
9       }
10      Gebe die Knotenmenge C aus;
11      V' := V' \ C;
12   }
```

sind alle Knoten in C paarweise in beiden Richtungen über Pfade verbunden und liegen deshalb zurecht in einer gemeinsamen Komponente. Es kann auch nicht sein, dass ein weiterer Knoten, der eigentlich in C liegen müsste, „vergessen“ wird. Denn ein solcher Knoten wäre mit Pfaden von und zu allen Knoten aus C verbunden, insbesondere also auch mit dem Knoten v aus Zeile 3. Aufgrund der Zeilen 5–7 wäre er dann auch in C aufgenommen worden. □

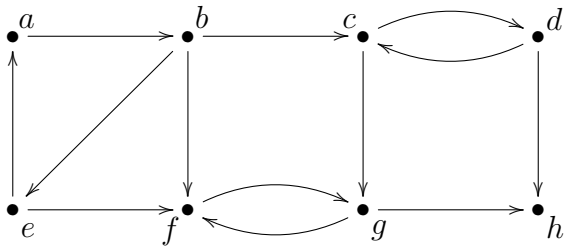
Leider aber ist der Algorithmus relativ langsam. Z.B. könnte der Graph tatsächlich aus nur einer einzigen starken Zusammenhangskomponente bestehen. In diesem Fall benötigt jede der n DFS–Durchmusterungen aus dem Vorbereitungsschritt $\Omega(n+m)$ viel Zeit, da jeweils alle Knoten und Kanten besucht werden. Allein für die Vorbereitung beträgt die Zeitkomplexität also schon $\Omega(n^2 + nm)$.

Überraschenderweise kann man das Problem aber auch viel schneller lösen. Der folgende Algorithmus berechnet erneut alle starken Zusammenhangskomponenten, benötigt dafür aber nur $O(n + m)$ viel Zeit. Im ersten Teil (Zeilen 1–11) ruft der Algorithmus wiederholt das DFS–Verfahren auf und sammelt alle markierten Knoten sortiert nach ihren *Pop*–Zeiten auf einem zusätzlichen Stack L. Im zweiten Teil (Zeilen 14–22) werden alle Knoten von L nacheinander wieder ausgelesen, um von dort aus jeweils erneut DFS–Durchmusterungen zu starten. Der Stack L dient lediglich dazu, die Reihenfolge aller Knoten aus dem ersten Schritt umzudrehen.

Zwischen dem ersten und dem zweiten Teil des Verfahrens werden zudem alle Kanten des Graphen umorientiert. Der Sinn dieser Massnahme wird später bei dem Korrektheitsbeweis deutlich werden.

```
ALGORITHMUS STARKE ZUSAMMENHANGSKOMPONENTEN

 1    for s ∈ V do {
 2       if (s ist noch nicht markiert) then {
 3          Führe den Algorithmus DFS mit s als Startknoten aus,
 4          lasse dabei aber die Uhr u von DFS immer weiter laufen
 5          (d.h Zeile 4 von DFS wird nur beim ersten DFS-Aufruf
 6          ausgeführt; danach wird die Anweisung durch u := u + 1
 7          ersetzt). Führe zudem bei jeder Pop(Q)-Anweisung in
 8          Zeile 19 von DFS zusätzlich die Operation Push(L, v)
 9          durch.
10       }
11    }
12    Entferne alle Knotenmarkierungen von G;
13    Drehe die Richtungen aller Kanten von G um;
14    while not Stack-empty(L) do {
15       s := top(L);
16       Pop(L);
17       if (s ist noch nicht markiert) then {
18          Rufe den Algorithmus DFS mit s als Startknoten auf und
19          gebe die Menge aller neu markierten Knoten als nächste
20          starke Zusammenhangskomponente aus.
21       }
22    }
```

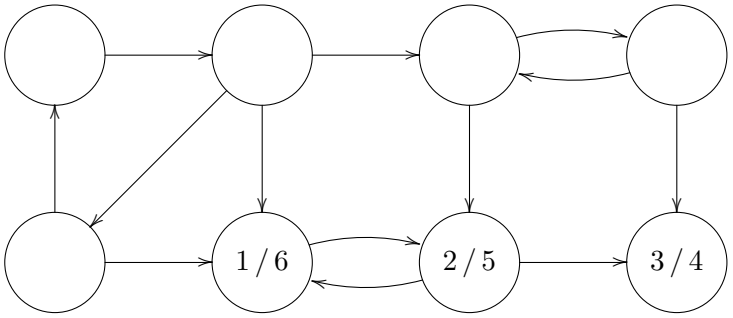
1.3.5 Beispiel Wir betrachten die Arbeitsweise des Verfahrens STARKE ZUSAMMENHANGSKOMPONENTEN an dem Graph $G = (V, E)$ aus Beispiel 1.2.1 mit der Knotenmenge $V = \{a, b, c, \ldots, h\}$.



Die Knotenmarkierungen $a, b, \ldots, h$ werden wir in den nachfolgenden Darstellungen aus Platzgründen wieder weglassen.

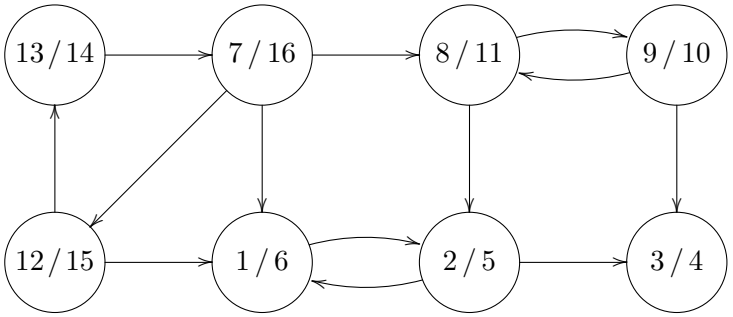
Angenommen, die `for`–Schleife in Zeile 1 durchläuft die Knotenmenge V in der Reihenfolge f, b, h, c, d, a, e, g (eine bestimmte Reihenfolge ist ja nicht vorgegeben). Dann

markiert der erste DFS–Aufruf mit f als Startknoten die drei unteren rechten Knoten f, g und h des Graphen. Für jeden Knoten v sind wieder wie im Beispiel 1.2.4 die *Push*– und *Pop*–Zeiten als Zahlenpaare $(d[v] \,/\, f[v])$ eingetragen. (Die *Push*–Zeiten dienen dabei nur dem Verständnis und sind für den Algorithmus selbst nicht wichtig.) Die *Push*– und *Pop*–Zeiten sind hier eindeutig, weil zuerst f, dann g, und abschließend h besucht wird. Alle anderen (leeren) Knoten werden nicht gefunden und sind von f aus nicht erreichbar.



Der erste Knoten, der von dem DFS–Keller entfernt wurde, war der rechte untere Knoten h, der zweite Knoten war g, und der dritte Knoten war f (denn es gilt $f[h] = 4$, $f[g] = 5$ und $f[f] = 6$). In dem Keller L befindet sich deshalb momentan die Knotensequenz $[h, g, f]$, wobei h der unterste und f den oberste Knoten des Kellerspeichers ist, d.h. nur auf f könnte aktuell zugegriffen werden.

Der nächste Knoten in der angenommen Reihenfolge f, b, h, c, d, a, e, g ist b. Dieser Knoten ist bislang noch unmarkiert und löst deshalb die nächste DFS–Durchmusterung aus. Hierbei werden alle restlichen Knoten gefunden. Es gibt diesmal zwei Möglichkeiten für die Vergabe der *Push*– und *Pop*–Zeiten, je nachdem, ob ausgehend von b zuerst c und später dann e besucht wird oder umgekehrt erst e und dann c. Wir gehen hier vom ersten Fall aus und erhalten die folgende Situation.

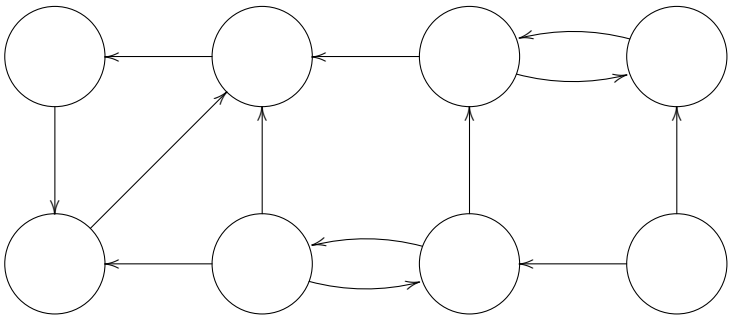


Beachten Sie, dass diesmal die Uhr u beim zweiten DFS–Aufruf weiterläuft und deshalb die erste *Push*–Zeit nicht 1, sondern 7 ist. Der Inhalt des Kellers L ist jetzt $[h, g, f, d, c, a, e, b]$, wobei b das oberste Element mit der größten *Pop*–Zeit darstellt.

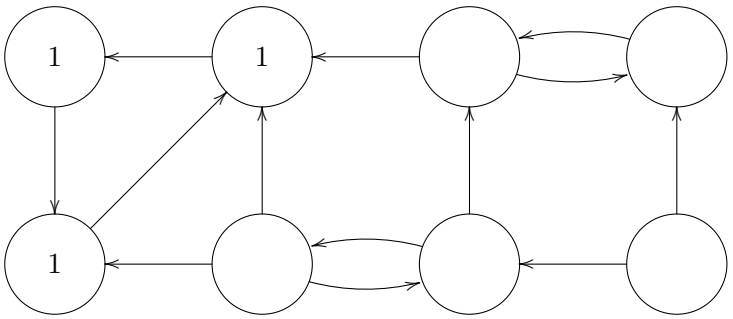
Von der angenommenen Knotenreihenfolge f, b, h, c, d, a, e, g bei den Durchläufen der `for`–Schleife in Zeile 1 sind jetzt noch die letzten sechs Knoten h, c, d, a, e, g übrig. Alle diese Knoten sind jedoch bereits markiert, d.h. die Bedingung in Zeile 2 ist jeweils nicht

erfüllt, und die restliche `for`–Schleife bleibt deshalb ohne weitere Auswirkungen. Der erste Teil des Algorithmus ist damit beendet.

Nun werden in den Zeilen 12 und 13 alle Kanten umorientiert und die Markierungen entfernt:



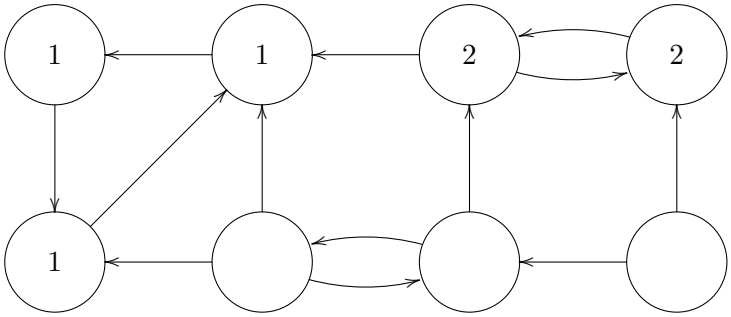
Im zweiten Teil des Algorithmus (Zeilen 14–22) wird anschließend jeder Knoten vom Keller L entfernt. Sofern ein solcher Knoten noch unmarkiert ist, wird erneut ein DFS–Verfahren von dort aus gestartet. Der oberste Knoten in L ist b, und b ist (wie alle anderen Knoten auch) momentan noch unmarkiert. Der entsprechende DFS–Aufruf mit b als Startknoten markiert die drei Knoten b, a, e (in dieser Reihenfolge) und gibt sie als erste starke Zusammenhangskomponente $\{a, b, e\}$ bekannt. Es kommt jetzt nicht mehr auf die *Push*– und *Pop*–Zeiten an, d.h. wir kennzeichnen die belegten Knoten einfach z.B. mit dem Zahlensymbol „1“:



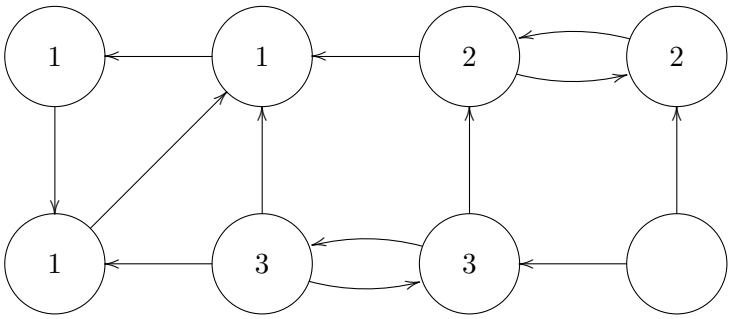
Nach der Entnahme von b besteht der Inhalt des Kellers L noch aus

$$[h, g, f, d, c, a, e] \ .$$

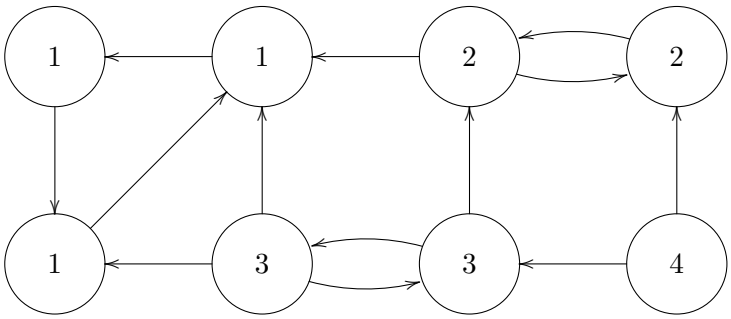
Die Knoten e und a sind bereits markiert und werden deshalb ohne weitere Auswirkungen von L entfernt. Erst beim Knoten c wird das nächste DFS–Verfahren mit c als Startknoten angestoßen. Beachten Sie, dass zwar z.B. der Knoten b von c aus erreichbar ist, dieser aber bereits markiert wurde. Die Tiefensuche von c aus führt deshalb nur noch zu einer Markierung der Knoten c und d, d.h. als zweite starke Zusammenhangskomponente wird $\{c, d\}$ ausgegeben. Wir benutzen die „2“ zur Kennzeichnung dieser Komponente.



Jetzt steht im Keller L noch die Knotenfolge $[h, g, f, d]$. Der Knoten d ist bereits markiert und wird folgenlos entnommen. Die Entnahme von f führt zu einem neuen DFS–Aufruf und zur Markierung der beiden Knoten f und g:



L besteht nun nur noch aus dem Inhalt $[h, g]$. Die Entnahme von g ist wieder wirkungslos, während h einen letzten neuen DFS–Aufruf auslöst, der allerdings nur noch h selbst markiert:



Damit endet das Verfahren. Ein Vergleich mit Beispiel 1.3.3 auf Seite 28 zeigt, dass alle starken Zusammenhangskomponenten korrekt identifiziert wurden.

Vor dem etwas kniffligen Korrektheitsbeweis analysieren wir zuerst die Laufzeit des Verfahrens.

1.3.6 Satz Der Algorithmus Starke Zusammenhangskomponenten benötigt für die Ausführung nur $O(|V| + |E|)$ viel Zeit.

Beweis: Angenommen, die DFS–Aufrufe in den Zeilen 3–9 würden fehlen. Dann benötigt die `for`–Schleife in den Zeilen 1–11 sicherlich nur $O(|V|)$ viel Zeit. Seien jetzt $V_1, V_2, V_3, \ldots, V_k$ die durch die DFS–Aufrufe (Zeilen 3–9) markierten Knotenmengen. Wie im Beweis von Satz 1.2.3 benötigen die Aufrufe dafür jeweils

$$O\Big(\sum_{v \in V_i} (outdeg(v) + 1)\Big)$$

viel Zeit ($i = 1, \ldots, k$). Beachten Sie, dass die DFS–Aufrufe immer nur disjunkte Teile des Graphen durchmustern (alles, was in vorherigen Aufrufen bereits markiert wurde, wird nicht nochmals durchmustert). Also sind die V_i–Mengen paarweise disjunkt. Andererseits wird aber auch jeder Knoten durch einen DFS–Aufruf abgedeckt, d.h. es gilt

$$\bigcup_{i=1}^{k} V_i = V \ .$$

Der Gesamtaufwand aller DFS–Aufrufe lässt sich somit zu

$$O\Big(\sum_{i=1}^{k} \sum_{v \in V_i} (outdeg(v) + 1)\Big) = O\Big(\sum_{v \in V} (outdeg(v) + 1)\Big) = O(|E| + |V|)$$

zusammenfassen. Folglich benötigt auch der komplette erste Teil nur

$$O(|V|) + O(|E| + |V|) = O(|V| + |E|)$$

viel Zeit, und für den zweiten Teil (Zeilen 14–22) greifen genau die gleichen Argumente.

In den Zeilen 12 und 13 muss aus dem Graph G ein neuer, unmarkierter Graph mit umorientierten Kanten erzeugt werden. Hierzu durchlaufen wir alle Adjazenzlisten ℓ_v von G und erzeugen für jeden Eintrag $w \in \ell_v$ einen neuen Eintrag v in der entsprechenden neuen Adjazenzliste ℓ_w von G'. Da alle Adjazenzlisten nur $O(|E| + |V|)$ viel Speicher belegen, können wir diese auch problemlos in der gleichen Zeit durchlaufen und die neuen Listen anlegen.

Die Zeitkomplexität des gesamten Algorithmus lässt sich somit auf

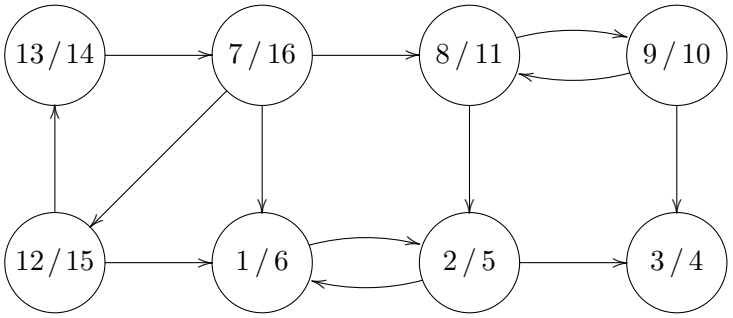
$$3 \cdot O(|V| + |E|) = O(|V| + |E|)$$

begrenzen. □

1.3.7 Definition Während der ersten Phase des Algorithmus Starke Zusammenhangskomponenten wird jeder Knoten $v \in V$ mit einer *Pop–Zeit* $f[v]$ versehen. Die *Pop–Zeit* $f(C)$ einer starken Zusammenhangskomponente $C \subseteq V$ bezeichnet den Zeitpunkt, an dem letztmalig ein Knoten aus C von einem DFS–Keller entfernt wird, d.h.

$$f(C) := \max\{f[v] \,|\, v \in C\} \ .$$

1.3.8 Beispiel Im Beispiel 1.3.5 wurde nach dem ersten Teil des Algorithmus die folgende Attributierung des Graphen mit *Push*– und *Pop*–Zeiten erreicht:



Somit gilt $f(\{a, b, e\}) = 16$, $f(\{c, d\}) = 11$, $f(\{f, g\}) = 6$, und $f(\{h\}) = 4$.

1.3.9 Satz Es seien $C, C' \subseteq V$ zwei verschiedene starke Zusammenhangskomponenten von G. Ferner seien zwei Knoten $v \in C$ und $v' \in C'$ durch eine Kante von v nach v' miteinander verbunden. Dann gilt:

- Es existiert kein Pfad von einem Knoten in C' zu einem Knoten in C.
- Nach dem Durchlauf der ersten Phase des Algorithmus Starke Zusammenhangskomponenten (Zeilen 1–11) gilt $f(C) > f(C')$.

Beweis: Bezüglich der ersten Behauptung nehmen wir an, ein solcher Pfad von einem Knoten $w' \in C'$ zu einem Knoten $w \in C$ würde existieren. Dann können wir wie folgt einen Pfad von v' nach v konstruieren:

- Es gibt einen Pfad von v' nach w', denn v' und w' liegen beide in C', und C' ist eine starke Zusammenhangskomponente, d.h. alle Knoten von C' sind durch Pfade in beide Richtungen miteinander verbunden, insbesondere also auch v' und w'.
- Anschließend verwenden wir den angenommenen Pfad von w' nach w.
- Die Knoten w und v liegen beide in der starken Zusammenhangskomponente C, also existiert analog zum ersten Teilstück ein Pfad von w nach v.

Also können wir einen Pfad von v' über w' und weiter über w nach v zusammensetzen. Da es nach Voraussetzung aber auch eine Kante von v nach v' gibt, haben wir so einen Knoten gefunden (nämlich v'), der zwar außerhalb von C liegt, aber trotzdem mit einem Knoten von C (nämlich v) über Pfade in beiden Richtungen verbunden ist. Dies widerspricht der zweiten Eigenschaft einer starken Zusammenhangskomponente (siehe Def. 1.3.2).

Für die zweite Behauptung betrachten wir alle Knoten aus C und C' und interessieren uns dafür, welcher von diesen Knoten zuerst markiert wird. Wir unterscheiden zwei Fälle:

Fall 1: Der erste markierte Knoten u liegt in C. Alle Knoten innerhalb von C sind von u aus erreichbar, insbesondere auch v. Wegen der Kante (v, v') ist somit auch v' von u aus erreichbar. Da es zudem Pfade von v' zu allen anderen Knoten in der zugehörigen

starken Zusammenhangskomponente C' gibt, sind also im Endeffekt alle Knoten in $C \cup C'$ von u aus ereichbar. Von u abgesehen sind alle Knoten in $C \cup C'$ zum Zeitpunkt $d[u]$ noch unmarkiert, da u der zuerst markierte Knoten ist. Also folgt aus Satz 1.2.6, dass $f[x] < f[u]$ für alle Knoten $x \in (C \setminus \{u\}) \cup C'$ gilt, insbesondere also speziell für alle Knoten $x \in C'$. Wegen $u \in C$ gilt deshalb

$$f(C') = \max\{f[x] \,|\, x \in C'\} < f[u] \leq f(C) \ .$$

Also haben wir für den ersten Fall $f(C) > f(C')$ nachgewiesen.

Fall 2: Der erste markierte Knoten u liegt in C'. Der Knoten u wird im Rahmen einer DFS–Durchmusterung markiert. Alle von u aus erreichbaren Knoten werden in dem gleichen DFS–Lauf markiert, also mindestens alle Knoten von C', da diese zusammen mit u in einer gemeinsamen starken Zusammenhangskomponente liegen und zum Zeitpunkt $d[u]$ noch alle unmarkiert sind. In dem gleichen DFS–Lauf werden bis zur Entnahme von u vom DFS–Keller jedoch keine Knoten aus C markiert, denn sonst wäre nach Satz 1.2.6 ein solcher Knoten $x \in C$ von $u \in C'$ aus erreichbar, was gemäß der ersten Behauptung dieses Satzes unmöglich ist. Also werden die Knoten von C erst später markiert, d.h. für alle Knoten $x \in C$ und $y \in C'$ gilt $d[x] > f[y]$ und deshalb auch $f[x] > f[y]$. Daraus können wir auch hier auf $f(C) > f(C')$ schließen.

Somit sind beide Behauptungen bewiesen. □

Wir betrachten nun den Graph G', der aus G in Zeile 13 erzeugt wird, indem alle Kanten von G umorientiert werden. Formal handelt es sich bei G' also um den Graph (V, E') mit $E' := \{(w, v) \,|\, (v, w) \in E\}$. Wir halten zwei wichtige Eigenschaften von G' fest.

1.3.10 Korollar Seien C' und C zwei starke Zusammenhangskomponenten von G mit $f(C') > f(C)$. Dann gibt es in G' keine Kante $(v', v) \in E'$ von einem Knoten $v' \in C'$ zu einem Knoten $v \in C$.

Beweis: Falls doch, so entspricht dies im Originalgraph G einer Kante von $v \in C$ nach $v' \in C'$. Satz 1.3.9 zufolge würde dies jedoch $f(C) > f(C')$ implizieren. □

1.3.11 Satz Die starken Zusammenhangskomponenten von G und G' sind identisch.

Beweis: Genau dann wenn es zwischen zwei Knoten u und v in G einen Pfad von u nach v und einen Pfad von v nach u gibt, gibt es umgekehrt in G' je einen Pfad von v nach u und von u nach v. Im Endeffekt liegen also die gleichen Bedingungen für die Bildung von starken Zusammenhangskomponenten vor. □

1.3.12 Satz Der Algorithmus Starke Zusammenhangskomponenten arbeitet korrekt.

Beweis: Im dritten Teil des Algorithmus (Zeilen 14–22) wird jeder Knoten $s \in V$ untersucht und, bei fehlender Markierung, eine DFS–Durchmusterung mit s als Startknoten eingeleitet. Bekanntlicherweise wird dabei zumindest der Startknoten s selbst markiert. Folglich sind am Ende alle Knoten markiert, d.h. jeder Knoten wird auch genau einmal als Inhalt einer Komponente ausgegeben (siehe Zeilen 18–20). Der Algorithmus berechnet also auf jeden Fall eine vollständige Zerlegung der Knotenmenge V.

Wir nehmen jetzt an, dass eine oder mehrere Komponenten nicht stark zusammenhängend sind und betrachten die erste dieser falsch berechneten Komponenten. Sei s der Startknoten aus Zeile 18, mit dem das zugehörige DFS–Verfahren aufgerufen wurde. Die tatsächliche starke Zusammenhangskomponente C' von Graph G mit $s \in C'$ ist auch im Graph G' unverändert vorhanden (Satz 1.3.11). Also sind alle Knoten in C' von s aus erreichbar und tragen noch keine Markierung, da alle früheren Komponenten bislang korrekt berechnet wurden. Folglich werden zumindest alle Knoten in C' durch das DFS–Verfahren neu markiert. Wenn also die Komponente tatsächlich falsch berechnet wurde, so verbleibt nur die Möglichkeit, dass zusätzliche Knoten außerhalb von C' fälschlicherweise mit markiert worden sind und später zusammen mit C' als angebliche Komponente ausgegeben wurden. Unter all diesen zuviel markierten Knoten betrachten wir denjenigen Knoten $v \notin C'$, der als erstes zuviel markiert wurde. Dann handelt es sich um einen Knoten, der durch eine Kante $(v', v) \in E'$ von einem Knoten $v' \in C'$ aus erreichbar ist (wenn nicht, so wäre schon einer der Vorgängerknoten von v vorher falsch mit markiert worden, ein Widerspruch). Wir betrachten nun die starke Zusammenhangskomponente C, in der v in Wirklichkeit liegt.

Fall 1: Es gilt $f(C) > f(C')$. Sei w der zuletzt von einem DFS–Keller entfernte Knoten von C, d.h. w hat die größe *Pop*–Zeit. Dann ist w für den Wert $f(C)$ verantwortlich, d.h. es gilt $f[w] = f(C)$. Nun werden bekanntlicherweise aus dem Keller L in den Zeilen 15 und 16 nacheinander alle Knoten aus V entnommen, und zwar in einer absteigend nach den zugehörigen *Pop*–Zeiten sortierten Reihenfolge. Wegen $f[w] = f(C) > f(C')$ wird also w noch vor jedem anderen Knoten aus C' untersucht. Als das fragliche DFS–Verfahren mit s als Startknoten aufgerufen wurde, war demnach eine frühere DFS–Durchmusterung mit w als Startknoten bereits abgeschlossen. Die zugehörige Komponente C wurde korrekt ermittelt, da alle Komponenten vor C' richtig berechnet wurden. Also ist insbesondere der Knoten $v \in C$ schon längst markiert und kann somit nicht mehr erneut markiert werden.

Fall 2: Es gilt $f(C') > f(C)$. Dann kann gemäß Korollar 1.3.10 die Kante $(v', v) \in E'$ gar nicht existieren. Somit ist dieser Fall unmöglich.

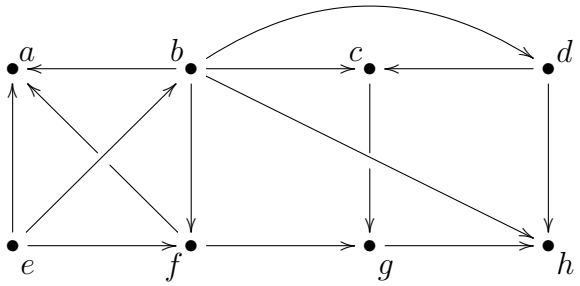
Es kann also keinen zusätzlichen falsch markierten Knoten $v \notin C'$ geben. Also arbeitet das fragliche DFS–Verfahren korrekt, d.h. die Komponente C' wird im Widerspruch zur Wahl von C' doch richtig berechnet. Also existieren keine falsch berechneten Komponenten. Die Korrektheit des Verfahrens ist damit nachgewiesen.

Es sei zum Abschluß bemerkt, dass die *Push*– und *Pop*–Zeiten für den Algorithmus selbst nicht benötigt werden, sondern nur für den Beweis. Auch die „Uhr“ u aus den Zeilen 4,

15 und 20 des DFS–Algorithmus ist somit überflüssig. Die entsprechenden Zeilen können entfernt und auch die *Push*– und *Pop*–Arrays d und f eingespart werden. □

1.4. Topologische Sortierungen

In vielen Situationen des täglichen Lebens können bestimmte Aufgaben erst dann erledigt werden, wenn zuvor andere Aufgaben erfolglich beendet worden sind. Denken Sie z.B. an die Produktion eines Autos. Ein Motor kann erst dann in den Motorraum eingesetzt werden, wenn zuvor die Karosserie geformt und fertig lackiert wurde. Solche Abhängigkeiten kann man gut durch einen gerichteten Graphen modellieren. Jeder Knoten repräsentiert dabei eine bestimmte Aufgabe, und eine Kante $(v, w) \in E$ macht deutlich, dass vor Erledigung der Aufgabe w erst die Aufgabe v erledigt werden muss. Insgesamt kann sich so ein Graph wie der folgende ergeben:



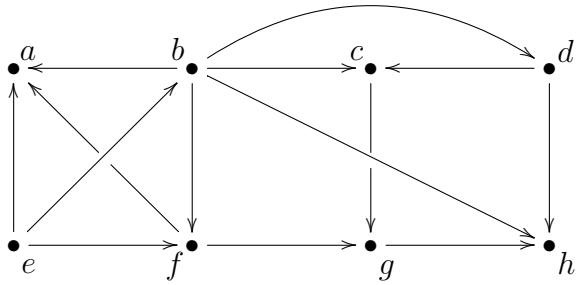
Natürlich ist der resultierende Graph $G = (V, E)$ immer kreisfrei, denn ansonsten könnte man bei keiner Aufgabe beginnen, die auf einem solchen Kreis liegt. Insbesondere muss es dann zumindest eine Aufgabe geben, die von überhaupt keiner anderen Aufgabe abhängt und bei der begonnen werden kann. Dies ist im obigen Graph bei dem Knoten e der Fall.

Wir möchten jetzt eine *topologischen Sortierung* $v_1, v_2, \ldots, v_n$ aller Knoten angeben, so dass die zugehörigen Aufgaben in dieser Reihenfolge abgearbeitet werden können. Eine Kante $(v_i, v_j) \in E$ impliziert also stets $i < j$.

Leider durchmustern weder DFS noch BFS die Knoten topologisch. Im obigen Graph könnte z.B. beim Aufruf von DFS mit e als Startknoten als nächster Knoten a markiert werden. Dies wäre aus topologischer Sicht unzulässig, da a z.B. von f abhängt. Bei BFS könnte genau dasselbe passieren. Zudem müsste man erst einmal wissen, dass man bei e anfangen soll. Wir benötigen also ein neues Verfahren, welches durch den folgenden Algorithmus beschrieben wird. Es verwendet ein Integer–Array *indeg*, welches für jeden Knoten die Anzahl der unmittelbaren Vorgängerknoten repräsentiert, die bislang noch nicht ausgegebenen wurden. Außerdem kommt wieder eine Knotenmenge Q im Programm vor. Sie enthält alle diejenigen Knoten, die als nächstes in die Sortierung eingehen dürfen, da alle Vorgänger schon ausgegeben wurden.

```
Algorithmus Topologische Sortierung

 1    for v ∈ V do {
 2       indeg[v] := 0;
 3    }
 4    for v ∈ V do {
 5       for w ∈ ℓ_v do { // ℓ_v = Menge der Nachfolgerknoten von v
 6          indeg[w] := indeg[w] + 1;
 7       }
 8    }
 9    Q := ∅;
10    for v ∈ V do {
11       if indeg[v] = 0 then {
12          Q := Q ∪ {v};
13       }
14    }
15    while Q ≠ ∅ do {
16       Wähle v ∈ Q;
17       Q := Q \ {v};
18       Gebe v als nächsten Knoten der sortierten Liste aus;
19       for w ∈ ℓ_v do {
20          indeg[w] := indeg[w] − 1;
21          if indeg[w] = 0 then {
22             Q := Q ∪ {w};
23          }
24       }
25    }
```

1.4.1 Beispiel Wir betrachten nochmals den obigen Beispielgraph:



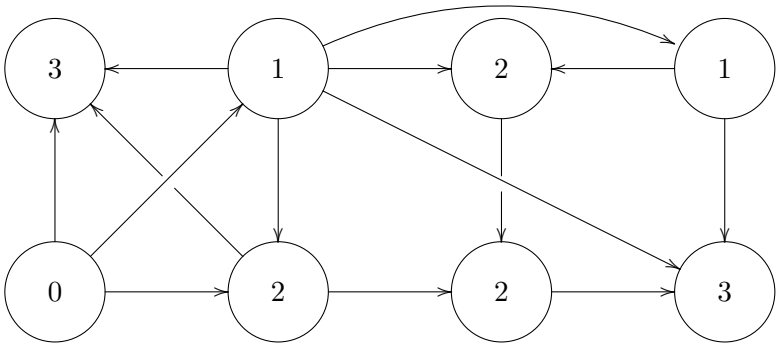
Wir notieren für jeden Knoten $v \in V$ den aktuellen Wert $indeg[v]$ in dem zugehörigen Kreis des Knotens. Wir geben zudem die in Q gespeicherte Menge an und notieren auch

die bis dahin erfolgte Ausgabe der topologisch sortierten Knotenfolge.

Es ist klar, dass während der ersten Phase des Algorithmus (Zeilen 1–9) das *indeg*–Array gerade so initialisiert wird, dass $indeg[v]$ dem Eingangsgrad von v für alle Knoten $v \in V$ entspricht.

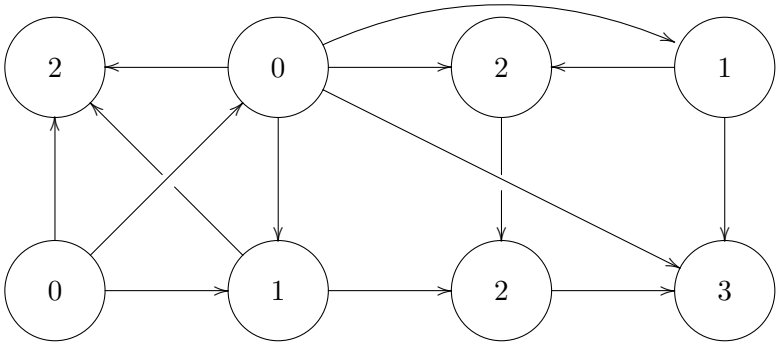
In unserem Beispiel liegt deshalb dann die folgende Situation vor:

$Q = \emptyset$ Bislang erfolgte Ausgabe: —



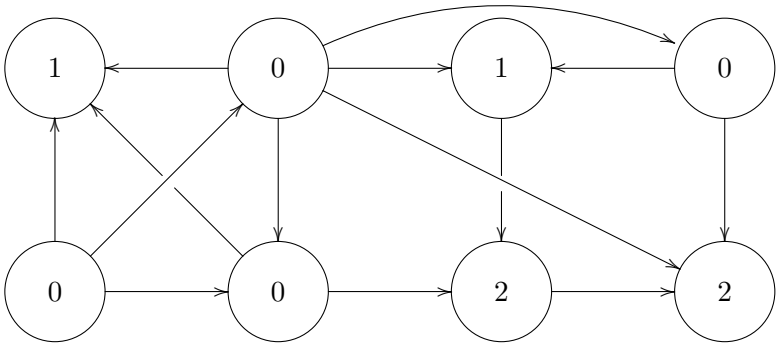
Folglich enthält Q nach der Ausführung der nächsten Zeilen 10–14 nur den Knoten e. Dieser Knoten wird anschließend in Zeile 16 ausgewählt, ausgegeben, und aus Q entfernt. Für alle Nachfolgerknoten a, b und f wird danach (Zeilen 19–24) der *indeg*–Wert einmal reduziert. Weil dann $indeg[b] = 0$ gilt, wird b neu in Q aufgenommen und bildet dort das einzige Element. Vor dem nächsten Durchlauf der `while`–Schleife lautet die aktuelle Situation also wie folgt:

$Q = \{b\}$ Bislang erfolgte Ausgabe: (e)



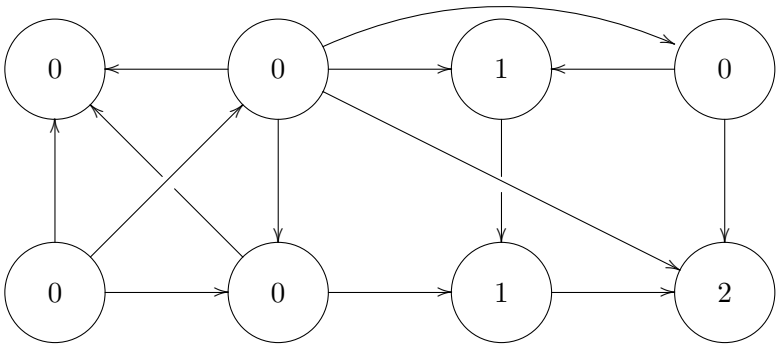
Nun wird b ausgewählt, ausgegeben, und die *indeg*–Werte der Nachfolgerknoten a, c, d, f, und h werden verringert. Also werden d und f in Q neu aufgenommen:

$Q = \{d, f\}$ Bislang erfolgte Ausgabe: (e, b)



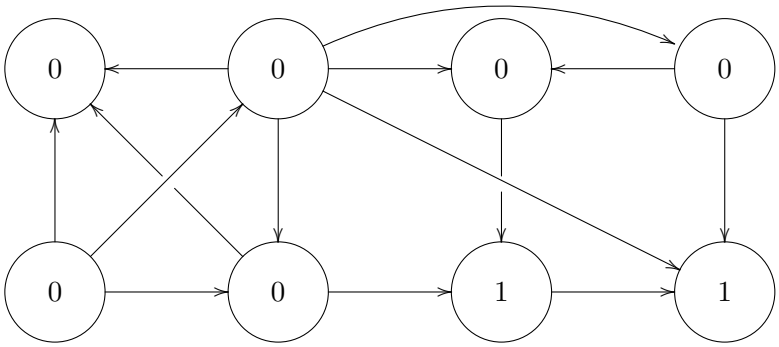
Jetzt besteht in Zeile 16 zum ersten Mal die Wahl zwischen zwei Knoten in Q. Angenommen, es wird der Knoten f ausgewählt. Die Verarbeitung von f ergibt:

$Q = \{a, d\}$ Bislang erfolgte Ausgabe: (e, b, f)



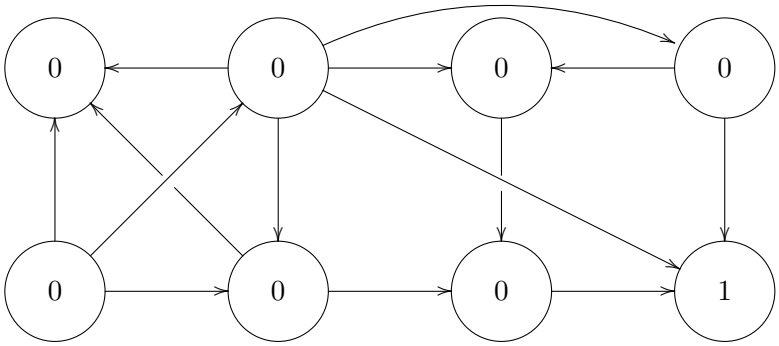
Falls nun a ausgewählt, ausgegeben, und aus Q entnommen wird, so bleibt die Verarbeitung der Zeilen 19–24 ohne Auswirkungen, da a keine Nachfolgerknoten hat. Die anschließende Entnahme und Ausgabe von d hingegen vermindert die *indeg*–Werte von c und h, und insbesondere wird c als neuer Knoten in Q aufgenommen:

$Q = \{c\}$ Bislang erfolgte Ausgabe: (e, b, f, a, d)



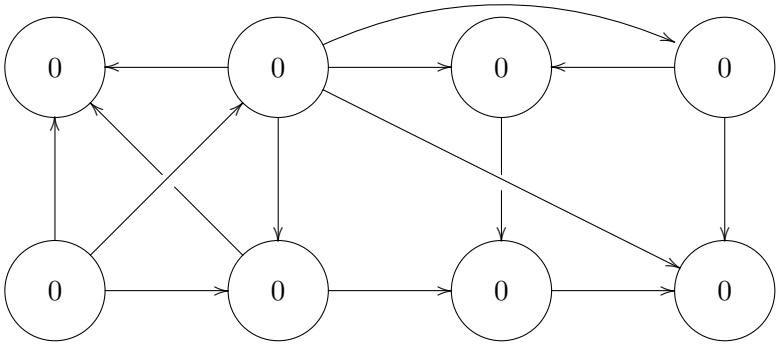
Die Verarbeitung von c führt zu der Aufnahme von g nach Q:

$Q = \{g\}$	Bislang erfolgte Ausgabe: (e, b, f, a, d, c)



Also wird nun g verarbeitet:

$Q = \{h\}$	Bislang erfolgte Ausgabe: (e, b, f, a, d, c, g)



Nun wird noch h ausgewählt, aber abgesehen von der Ausgabe dieses Knotens werden keine Veränderungen an den *indeg*–Werten mehr vorgenommen, da h keine ausgehenden Kanten besitzt. Also gilt dann:

$Q = \emptyset$	Bislang erfolgte Ausgabe: (e, b, f, a, d, c, g, h)

Insbesondere terminiert nun der Algorithmus. Man prüft leicht nach, dass die ausgegebene Knotenfolge wie gewünscht einer topologischen Sortierung entspricht.

1.4.2 Satz Der Algorithmus Topologische Sortierung arbeitet korrekt.

Beweis: Die Menge Q ist anfangs leer (Zeile 9). Ein Knoten v kann nur nach Q gelangen, wenn sein *indeg*–Wert gleich Null ist (Zeilen 11–13 und 21–23). Also wird jeder Knoten nur höchstens einmal ausgegeben. Denn dazu muss er in Q enthalten sein und wird anschließend aus Q entfernt (Zeilen 16–18). Eine erneute Aufnahme nach Q wäre nur in den Zeilen 21–23 möglich, da aber der *indeg*–Wert dabei zuvor weiter heruntergezählt wird (Zeile 20), kann die Gleichheit zur Null nie mehr erfüllt werden.

Wir zeigen jetzt, dass beim Erreichen des Kopfes der `while`–Schleife (Zeile 15) immer folgende Invariante gilt:

Für jeden Knoten $v \in Q$ wurden bereits alle unmittelbaren Vorgänger ausgegeben.

Es ist klar, dass während der ersten Phase des Algorithmus (Zeilen 1–8) alle *indeg*–Werte gerade so initialisiert werden, dass $indeg[v]$ dem Eingangsgrad von v für alle Knoten $v \in V$ entspricht. Alle Knoten, die in den Zeilen 9–14 nach Q gelangen, weisen also keine Vorgänger auf und erfüllen deshalb trivialerweise die behauptete Invariante. Wenn später ein weiterer Knoten w nach Q aufgenommen wird, so muss der $indeg[w]$–Wert genau dem Eingangsgrad von w entsprechend oft heruntergezählt worden sein (Zeile 20). Bei jedem dieser Zählschritte wird zugleich ein Vorgängerknoten von w mit ausgegeben. Da aber jeder Knoten nur einmal ausgegeben wird, sind folglich alle Vorgängerknoten von w ausgegeben worden, bevor w nach Q gelangt. Die Invariante bleibt also weiterhin gültig.

Da nur Knoten aus Q ausgegeben werden, folgt aus der Invariante unmittelbar die korrekte topologische Sortierung der ermittelten Knotenfolge. Wir müssen jetzt nur noch prüfen, dass am Ende auch wirklich jeder Knoten in der Folge enthalten ist.

Angenommen, am Ende wären einige Knoten übrig geblieben. Dann muss es darunter zumindest einen Knoten w geben, dessen Vorgängerknoten bereits ausgegeben wurden (ansonsten wäre der Graph im Widerspruch zur Voraussetzung zyklisch). Folglich ist w nicht nach Q gelangt, denn als solcher wäre er irgendwann in den Zeilen 16 und 17 ausgewählt und ausgegeben worden, da vorher der Algorithmus nicht terminiert (Zeile

15). Die Ausgabe aller Vorgängerknoten impliziert jedoch, dass $indeg[w]$ korrekt bis auf Null heruntergezählt wurde. Bei der Ausgabe des letzten Vorgängerknotens muss deshalb w doch in Q aufgenommen worden sein (siehe Zeilen 21–23). Dies ist ein Widerspruch.

Der Algorithmus Topologische Sortierung arbeitet also korrekt. □

1.4.3 Satz Der Algorithmus Topologische Sortierung besitzt eine Laufzeit von $O(|V| + |E|)$.

Beweis: Die Zeilen 1–3 und 9–14 werden in $O(|V|)$ viel Zeit durchlaufen. Da alle Adjazenzlisten insgesamt genau $|E|$ viele Einträge besitzen (jeder Eintrag korrespondiert bekanntlich zu genau einer Kante), wird auch für die Ausführung der Zeilen 4–8 nicht mehr als $O(|V| + |E|)$ viel Zeit benötigt. Gleiches trifft auf die Zeilen 15–25 zu, da jeder Knoten höchstens einmal nach Q gelangt und somit auch hier jede Adjazenzliste in den Zeilen 19–24 nur höchstens einmal durchlaufen wird. □

1.4.4 Korollar Ein Graph kann in linearer Zeit auf seine Kreisfreiheit getestet werden.

Beweis: Bei einem gerichteten Graph ruft man einfach den Algorithmus Topologische Sortierung auf und zählt mit, ob die ausgegebene Knotenfolge vollständig ist. Wir haben im Beweis von Satz 1.4.2 gesehen, dass fehlende Knoten auf einen Zyklus im Graphen hindeuten.

In einem ungerichteten Graph ermittelt man dagegen die Anzahl der Zusammenhangskomponenten und prüft anschließend das Kriterium aus Satz 1.1.11. Zum Ermitteln der Zusammenhangskomponenten kann man wie im ersten Teil von dem Algorithmus Starke Zusammenhangskomponenten (Zeilen 1–11) vorgehen und die DFS–Aufrufe mitzählen. □

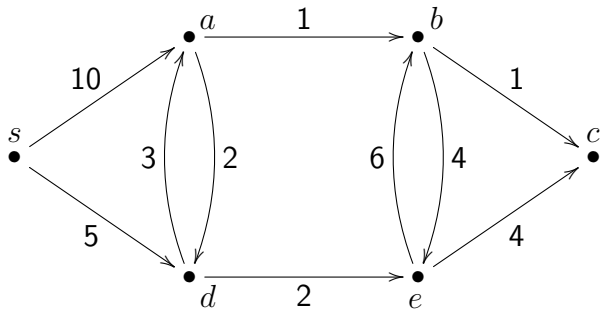
2 Kürzeste Wege

Ein häufiges Problem im Zusammenhang mit Graphen betrifft die Ermittlung von kürzesten Wegen. Jeder Routenplaner im Internet bzw. jedes Navigationsgerät im Auto ist heute in der Lage, die beste Verbindung zwischen zwei Adressen zu berechnen. Dieses Problem lässt sich leicht graphentheoretisch modellieren. Jede Straßenkreuzung wird durch einen Knoten, jede Straße durch eine Kante repräsentiert. Zusätzlich liegt eine *Gewichtsfunktion* $c : E \longrightarrow \mathbb{R}$ vor. Wenn später eine Kante $e \in E$ auf dem ausgewiesenen Pfad liegt, so werden dafür „Kosten“ i.H.v. $c(e)$ fällig. Es kann sich dabei um Mautgebühren, Entfernungen, benötigte Fahrzeiten, usw. handeln. Kosten können aber durchaus auch negativ sein. Ziel ist es, *kürzeste Pfade* zwischen Knoten in V zu ermitteln, bei denen die Gesamtkosten (= Summe der einzelnen Gewichte) minimal sind.

2.1. Grundlagen

2.1.1 Definition Ein *(kanten–)gewichteter Digraph* $G = (V, E, c)$ besteht aus einem gerichteten Graph (V, E) und einer *Gewichtsfunktion* $c : E \longrightarrow \mathbb{R}$.

2.1.2 Beispiel Im nachfolgenden Graph ist ein Startknoten s ausgezeichnet. Gesucht sind die kürzesten Pfade zwischen s und allen von s aus erreichbaren Knoten.



Für alle $v \in V$ bezeichne $k(v)$ die Länge eines kürzesten Pfades von s nach v. Wir vereinbaren ferner $k(v) = \infty$, falls v von s aus unerreichbar ist. In unserem Beispiel sind allerdings alle Knoten von s aus erreichbar. Die kürzesten Pfade und deren Längen lauten:

- von s nach a: $s \to d \to a$, $k(a) = 8$
- von s nach b: $s \to d \to a \to b$, $k(b) = 9$
- von s nach c: $s \to d \to a \to b \to c$, $k(c) = 10$
- von s nach d: $s \to d$, $k(d) = 5$
- von s nach e: $s \to d \to e$, $k(e) = 7$

Für alle erreichbaren Knoten $v \in V \setminus \{s\}$ bezeichnen wir weiter mit $p(v)$ den Vorgängerknoten von v auf einem kürzesten Pfad von s nach v. Folglich gilt hier

$$p(a) = d \ , \quad p(b) = a \ , \quad p(c) = b \ , \quad p(d) = s \quad \text{und} \quad p(e) = d \ .$$

Man kann einen kürzesten Pfad aus diesen Vorgängerwerten problemlos rekonstruieren (allerdings in umgekehrter Reihenfolge), indem man die Knotenfolge v, $p(v)$, $p(p(v))$, usw. ausgibt, bis man auf den Knoten s stösst. Für einen kürzesten Pfad von s nach b gilt beispielsweise

$$b \ , \quad p(b) = a \ , \quad p(p(b)) = p(a) = d \ , \quad p(p(p(b))) = p(d) = s \ ,$$

woraus wir den kürzesten Pfad

$$s \to d \to a \to b$$

ablesen können.

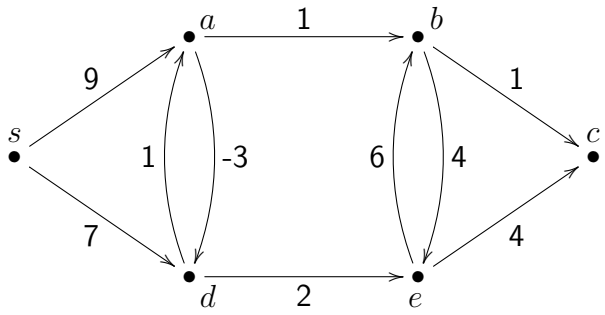
Kürzeste Pfade lassen sich nur dann sinnvoll berechnen, wenn keine sogenannten *negativen Kreise* in G vorliegen.

2.1.3 Definition Ein *negativer Kreis* (oder *negativer Zyklus*) ist ein Zyklus in G, für den die Summe aller Kantengewichte insgesamt kleiner als Null ist.

2.1.4 Beispiel Der nachfolgende Graph besitzt einen negativen Kreis der Länge

$$(-3) + 1 = -2$$

auf dem geschlossenen Pfad $a \to d \to a$.



Nur für Graphen ohne negative Kreise gelten die beiden folgenden Eigenschaften. Sie ermöglichen eine effiziente Suche nach kürzesten Pfaden.

2.1.5 Lemma Sei $G = (V, E, c)$ ein gewichteter Digraph ohne negative Zyklen. Dann gilt:

a) Jeder kürzeste Verbindung von s zu einem von s aus erreichbaren Knoten kann immer kreisfrei gewählt werden, d.h. es handelt sich wirklich um einen Pfad (zur Erinnerung: bei Def. 1.1.1 auf Seite 9 haben wir verlangt, dass Wege und Pfade keine doppelten Knoten enthalten dürfen).

b) Für alle erreichbaren Knoten $v \in V \setminus \{s\}$ gilt $k(v) = k(p(v)) + c((p(v), v))$.

Beweis: Jeder Kreis auf einer Verbindung ist nach Voraussetzung nicht–negativ und kann deshalb herausgeschnitten werden, ohne die Verbindung zu verlängern. Da die Verbindung schon vorher eine kürzeste war, kann übrigens jeder Kreis (wenn überhaupt) nur die neutrale Länge Null besitzen.

Bzgl. der zweiten Behauptung gilt zunächst immer $k(v) \leq k(p(v)) + c((p(v), v))$, denn eine Möglichkeit für einen Pfad von s nach v besteht gerade in einem Pfad von s nach $p(v)$, gefolgt von der abschließenden Kante $(p(v), v)$. Dieser Pfad lässt sich mit

$$k(p(v)) + c((p(v), v))$$

Kosten realisieren. Also kann der kürzeste Pfad von s nach v nicht länger sein. Wäre er kürzer, so würde aus $k(v) < k(p(v)) + c((p(v), v))$ auch $k(v) - c((p(v), v)) < k(p(v))$ folgen. Die linke Seite gibt die Kosten eines Pfades von s nach v ohne die abschließende Kante von $p(v)$ nach v wieder, also die Kosten eines Pfades von s nach $p(v)$. Diese Kosten wären dann geringer als $k(p(v))$, ein Widerspruch. □

Im Beispiel 2.1.4 ist auf Grund des negativen Kreises die erste Eigenschaft nicht erfüllt. Z.B. können wir den kreisfreien Pfad $s \to d \to a \to b$ der Länge 9 verkürzen, indem wir eine „Extrarunde“ zwischen a und d einlegen:

$$s \to d \to a \to d \to a \to b \ .$$

Diese Verbindung besitzt nur noch die Länge 7. Natürlich können wir noch weitere Schleifen auf dem Kreis durchlaufen und so die Kosten beliebig drücken. Es gibt also

keine wirklich „kürzeste“ Verbindung mehr. Man muss also — im Gegensatz zu der ersten Behauptung aus Lemma 2.1.5 — explizit verlangen, dass die kürzeste Verbindung kreisfrei (also ein Pfad) sein soll.

Im Beispiel 2.1.4 kann man sehen, dass auch die zweite Behauptung von Lemma 2.1.5 bei Graphen mit negativen Kreisen falsch ist. Der kürzeste Pfad von s nach d ist

$$s \ \rightarrow \ a \ \rightarrow \ d$$

und hat die Länge 6, und der kürzeste Pfad von s nach a ist

$$s \ \rightarrow \ d \ \rightarrow \ a$$

mit der Länge 8. Wegen $p(a) = d$ und

$$k(a) = 8 > 6 + 1 = k(d) + c((d, a)) = k(p(a)) + c((p(a), a))$$

gilt die behauptete Gleichheit

$$k(a) = k(p(a)) + c((p(a), a))$$

hier also nicht. Dies macht die Suche nach kürzesten Pfaden in solchen Graphen sehr schwierig, weil sich solche Pfade nicht notwendigerweise aus Teilstücken zusammensetzen lassen, die selbst kürzeste Pfade auf den entsprechenden Teilstrecken darstellen. Bis heute kennt man für Graphen mit negativen Kreisen nur Algorithmen mit exponentiellem Aufwand, die das Kürzeste–Wege–Problem lösen.

Wir setzen daher von jetzt an voraus, dass alle Graphen frei von negativen Kreisen sind. In den nachfolgenden Abschnitten stellen wir für solche Graphen zwei verschiedene Lösungsansätze vor.

2.2. Der Algorithmus von Dijkstra

Es sei $G = (V, E, c)$ ein gewichteter Digraph mit einer Gewichtsfunktion $c : E \rightarrow \mathbb{R}_0^+$. Die Gewichte werden hier also als nicht–negativ vorausgesetzt, was in der Praxis häufig, aber nicht immer erfüllt ist. Wie im Beispiel 2.1.2 werden die kürzesten Pfade zwischen s und allen von s aus erreichbaren Knoten gesucht.

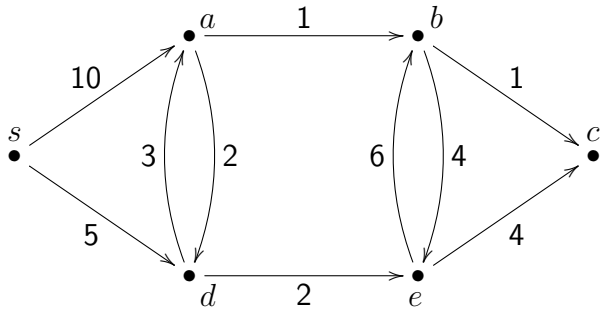
1959 entwickelte Dijkstra[1] für diese Problemstellung einen interessanten Algorithmus. Dieser verwaltet für jeden Knoten $v \in V$ einen Wert $k[v]$, der am Ende genau der kürzesten Weglänge $k(v)$ von s nach v entspricht. Außerdem wird für jeden von s aus erreichbaren Knoten $v \in V \setminus \{s\}$ ein Knoten $p[v]$ ermittelt, nämlich gerade der Vorgängerknoten $p(v)$ auf einem kürzesten Pfad von s nach v.

[1] Edsger Wybe Dijkstra, *1930 Rotterdam, †2002, Nuenen, Niederlande, niederländischer Informatiker, 1984–1999 Professor an der University of Texas at Austin.

Der Algorithmus verwendet wieder eine Knotenmenge Q, die bei jedem Durchlauf der Hauptschleife um ein Element schrumpft. Für alle Knoten außerhalb von Q sind die kürzesten Wege dann bereits bekannt. Aus Q werden nach und nach diejenigen Knoten entnommen, die unter allen verbliebenen Knoten in Q mit den geringsten Kosten von s aus erreichbar sind.

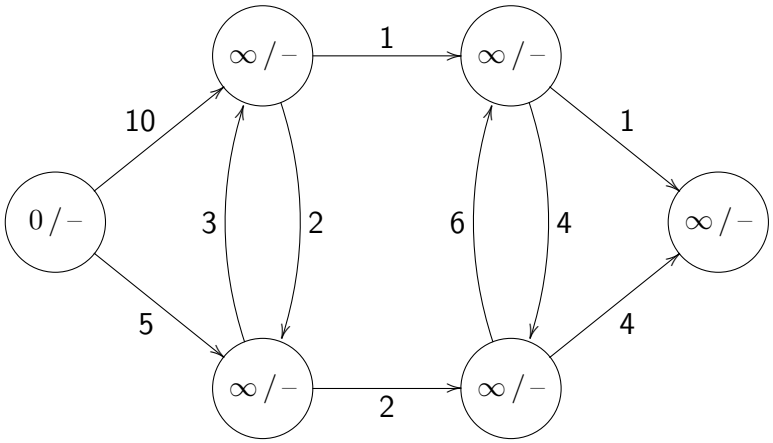
```
ALGORITHMUS VON DIJKSTRA

1    for v ∈ V \ {s} do {
2       k[v] := ∞;
3    }
4    k[s] := 0;
5    Q := V;
6    while Q ≠ ∅ do {
7       Wähle v ∈ Q, so dass k[v] minimal ist;
8       Q := Q \ {v};
9       for w ∈ ℓ_v do { // ℓ_v = Menge der Nachfolgerknoten von v
10         if w ∈ Q then {
11            if k[v] + c((v,w)) < k[w] then {
12               k[w] := k[v] + c((v,w));
13               p[w] := v;
14            }
15         }
16      }
17   }
```

2.2.1 Beispiel Wir wenden den Algorithmus von DIJKSTRA auf den obigen Beispielgraph an:



Wir notieren für jeden Knoten $v \in V$ die aktuelle Distanz $k[v]$ sowie den aktuellen Vorgängerknoten $p[v]$ in dem zugehörigen Kreis des Knotens als Paar $k[v]/p[v]$.

Nach der Verarbeitung der Initialisierung durch die Zeilen 1–5 bietet sich demnach folgendes Bild:



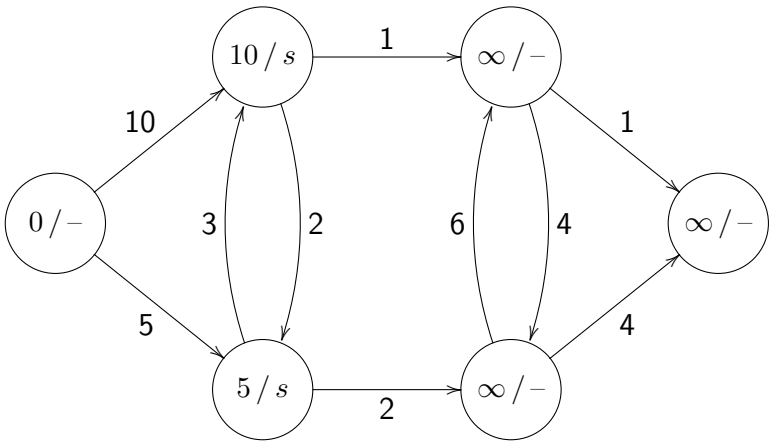
Die Knotenmenge Q umfasst anfangs ganz V (Zeile 5). Die Bedingung der `while`–Schleife ist also erfüllt, und in Zeile 7 wird dann ein Knoten mit minimalem k–Wert ausgewählt. Es gibt nur einen Knoten, der einen Wert kleiner als unendlich aufweist, nämlich den Startknoten s selbst. Also wird s ausgewählt und aus Q entfernt (Zeile 8). Anschließend werden alle Nachfolgerknoten von s (also a und d) in der `for`–Schleife (Zeilen 9–16) durchlaufen. Da Q momentan noch alle Knoten außer s enthält, ist beidesmal die Bedingung in Zeile 10 erfüllt. Gleiches gilt für die Bedingung in Zeile 11, denn zum einen gilt $k[s] = 0$, und zum anderen enthalten sowohl $k[a]$ als auch $k[d]$ noch den Wert unendlich. Geprüft werden in Zeile 11 also die Bedingungen

$$k[s] + c((s, a)) = 0 + 10 = 10 < \infty$$

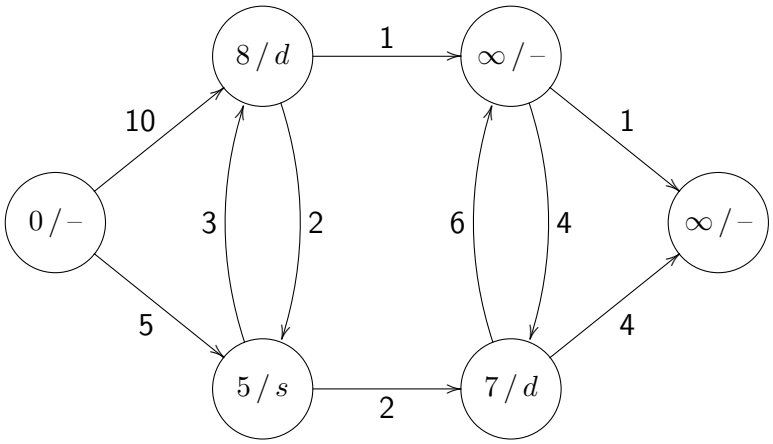
sowie

$$k[s] + c((s, d)) = 0 + 5 = 5 < \infty \ .$$

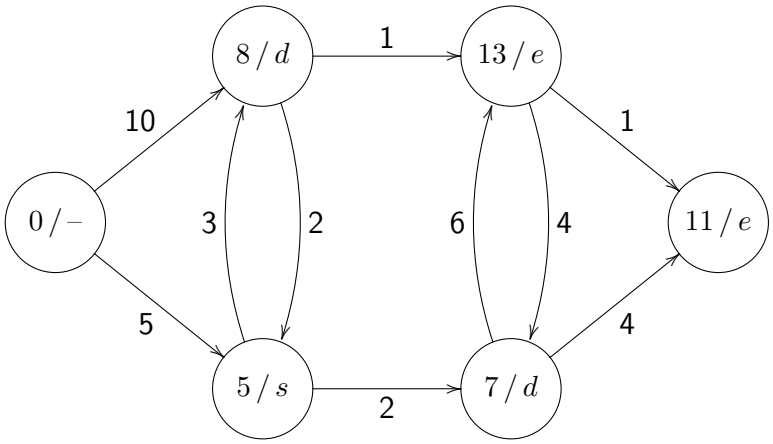
Beidesmal werden also die Zeilen 12 und 13 durchlaufen. Wir erhalten:



Q enthält jetzt noch alle Knoten außer s, also $\{a, b, c, d, e\}$. Unter diesen hat d den geringsten k–Wert, nämlich 5. Also wird im nächsten Durchlauf der `while`–Schleife nun d in Zeile 7 ausgewählt und in Zeile 8 aus Q entfernt. Beide Nachfolger a und e sind noch in Q vorhanden, und die Zeilen 12 und 13 führen erneut zu Verbesserungen der jeweiligen k–Werte. Konkret wird $k[a]$ auf $5 + 3 = 8$ reduziert, und $k[e]$ auf $5 + 2 = 7$. Aktuell ergibt sich so:



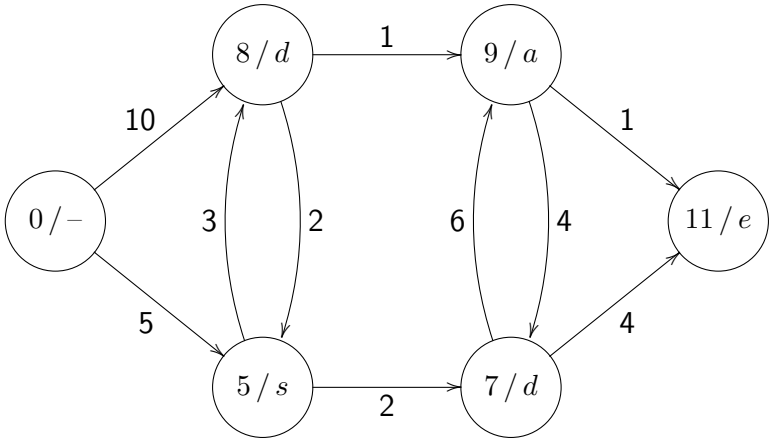
Jetzt gilt $Q = \{a, b, c, e\}$, und der Knoten mit dem geringsten k–Wert (nämlich 7) ist e. Die Nachfolger von e sind b und c, so dass auch deren k–Werte nun erstmals kleiner werden:



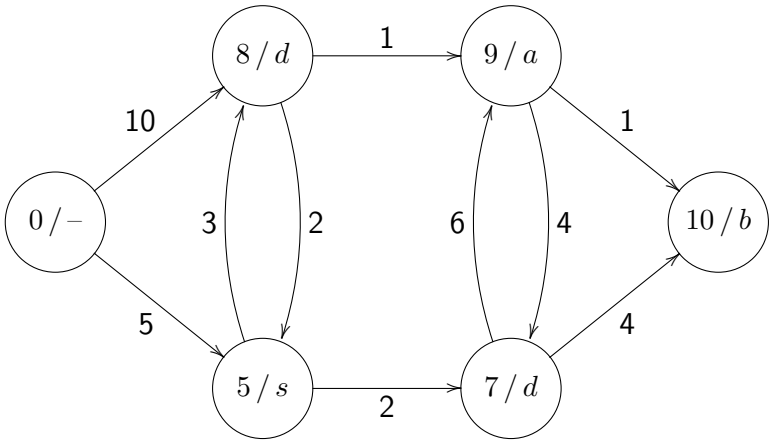
Beim nächsten Erreichen des `while`–Schleifenkopfes gilt $Q = \{a, b, c\}$, und a besitzt den geringsten k–Wert, nämlich 8. Die Nachfolgermenge ℓ_a umfasst b und d, aber nur b ist noch in Q vorhanden. Wegen der Bedingung in Zeile 10 bleibt die Untersuchung von d also ohne Auswirkungen. Die Zeilen 11–14 erzielen jedoch für $w = b$ eine Verbesserung für $k[b]$ auf den Wert

$$k[b] = k[a] + c((a, b)) = 8 + 1 = 9 \ .$$

Somit gilt jetzt:



Unter den beiden verbliebenen Knoten b und c in der Menge Q wird jetzt wegen $k[b] = 9$ und $k[c] = 11$ der Knoten b ausgewählt und aus Q entfernt. Der eine Nachfolgerknoten e ist nicht mehr in Q vorhanden, aber für den anderen Nachfolgerknoten c ergibt noch einmal eine Verbesserung auf den Wert $k[c] = 9 + 1 = 10$. Wir erhalten:



Nun enthält Q nur noch den Knoten c. Dieser wird nur noch ausgewählt und aus Q gelöscht. Er erzielt ansonsten keine Wirkung mehr, da kein einziger Knoten in Q mehr vertreten ist. Somit terminiert anschließend auch das Verfahren (Zeile 6).

Wir können leicht verifizieren, dass alle k–Werte (also die Distanzen) sowie alle Vorgängerknoten (also die p–Knoteneinträge) korrekt bestimmt wurden.

2.2.2 Satz Der Algorithmus von DIJKSTRA arbeitet korrekt.

Beweis: Wir zeigen durch Induktion über die Anzahl i der Schleifendurchläufe der `while`–Schleife, dass beim Erreichen von Zeile 6 immer folgendes gilt:

a) Für alle $v \in Q$ enthält $k[v]$ die Kosten eines kürzesten Pfads von s nach v, wobei (von v abgesehen) nur Knoten aus $V \setminus Q$ benutzt werden dürfen. Für unter diesen Umständen unerreichbare Knoten gilt $k[v] = \infty$. Im Fall $v \neq s$ und $k[v] < \infty$

enthält $p[v]$ den Vorgängerknoten von v auf einem solchen Pfad.

b) Für alle $v \in V \setminus Q$ enthält $k[v]$ die Kosten eines kürzesten Pfads von s nach v ohne irgendwelche Restriktionen, d.h. die ermittelten Kosten sind bereits optimal. Auch hier enthält $p[v]$ im Fall $v \neq s$ und $k[v] < \infty$ den entsprechenden Vorgängerverweis zur Rekonstruktion eines kürzesten Pfades.

$\underline{i = 0}$: Wenn Zeile 6 zum ersten Mal erreicht wird, gilt $Q = V$. Dann ist es für alle Knoten $v \in V \setminus \{s\}$ unmöglich einen Pfad von s nach v zu bilden. Denn alle Knoten des Pfades (abgesehen von v) müßten in $V \setminus Q$ liegen, also z.B. der Startknoten s. Wegen $Q = V$, also $V \setminus Q = \emptyset$, ist dies unmöglich. Für den Fall $v = s$ existiert dagegen der triviale Pfad von s nach s, und dieser hat die Kosten 0. Alle anderen Knoten sind unerreichbar (Kosten ∞), d.h. die in den Zeilen 1–4 zugewiesenen Werte sind korrekt. Natürlich ist auch die zweite Invariante wegen $V \setminus Q = \emptyset$ trivialerweise richtig.

$\underline{i \to i+1}$: Die beiden Invarianten seien bislang wahr, und es wird nun zum $(i+1)$–ten Mal der Rumpf der `while`–Schleife ausgeführt. Aus Q wird in Zeile 7 der Knoten v mit den geringsten Wegkosten ausgewählt. Diese Kosten können nicht mehr gesenkt werden. Ist nämlich

$$s = w_0 \ \to \ w_1 \ \to \ \ldots \ \to \ w_k = v$$

ein beliebiger Pfad von s nach v, so betrachte man zum jetzigen Zeitpunkt (wir befinden uns immer noch in Zeile 7) den ersten Knoten w_j mit $w_j \in Q$. Der Knoten w_j existiert wegen $w_k = v \in Q$. Ferner gilt

$$s = w_0, w_1, \ldots, w_{j-1} \in V \setminus Q \ .$$

Die erste Invariante besagt, dass die Kosten dieses Pfades von s nach w_j mindestens $k[w_j]$ betragen. Wegen der nicht–negativen Kantengewichte betragen die Gesamtkosten des Pfades

$$s = w_0 \ \to \ w_1 \ \to \ \ldots \ \to \ w_k = v$$

also ebenfalls mindestens $k[w_j]$. Nach der Wahl von v gilt aber $k[v] \leq k[w_j]$, d.h. der betrachtete Weg führt zu keiner besseren Lösung. Der Knoten v wird also zu Recht aus Q entlassen (Zeile 8) und befindet sich danach in $V \setminus Q$. Die zweite Invariante ist dann immer noch erfüllt.

Nun muss noch gezeigt werden, dass auch die erste Invariante weiterhin gilt. Für einen in der ersten Invariante beschriebenen Pfad von s nach w eröffnet sich die neue Möglichkeit, irgendwo den neu hinzu gekommenen Knoten v gewinnbringend einzusetzen. Betrachten wir also einen beliebigen Pfad von s über v nach w, wobei bis auf w alle Knoten aus $V \setminus Q$ stammen. Falls v dann nicht der vorletzte Knoten auf diesem Pfad ist, so ist der Nachfolger von v also ebenfalls in $V \setminus Q$ enthalten. Dann aber kann keine Verbesserung eintreten, denn allein für das Teilstück bis zu diesem Nachfolgerknoten haben wir gemäß der zweiten Invariante die kürzesten Wegkosten bereits ermittelt. Eine Verbesserung für einen Pfad von s nach w ist also nur nur dann möglich, wenn v der vorletzte Knoten (also der Vorgänger von w) ist. Wir müssen also zunächst von s nach v gehen, und abschließend die Kante (v, w) benutzen (sofern sie existiert). Die Kosten für einen solchen

Pfad betragen mindestens $k[v] + c((v, w))$. Falls dies eine Verbesserung darstellt, werden entsprechend $k[w]$ und $p[w]$ geändert (siehe Zeilen 11–14). Also sind nach dem Durchlauf des kompletten `while`–Schleifenrumpfes beide Invarianten weiterhin wahr.

Am Ende gilt $Q = \emptyset$, d.h. die zweite Invariante impliziert wegen $V \setminus Q = V$ die korrekte Berechnung aller Distanzen und Vorgängerknoten. □

2.2.3 Satz Der Algorithmus von DIJKSTRA benötigt eine Laufzeit von $O(n^2)$, wobei n die Kardinalität der Knotenmenge V bezeichnet.

Beweis: Die `while`–Schleife wird genau n–mal ausgeführt, da bei jedem Durchlauf ein Knoten v aus Q entnommen wird. Wenn wir für einen Moment die Zeile 7 von dem Verfahren ausblenden, so besteht der restliche Schleifenrumpf im wesentlichen aus der Durchmusterung der jeweiligen Adjazenzliste ℓ_v. Die Ausführungszeit der gesamten `while`–Schleife ist also (von Zeile 7 abgesehen) proportional zur Länge aller Adjazenzlisten, also $O(|V| + |E|)$. Auch die Initialisierung in den Zeilen 1–5 benötigt nur $O(n)$ viel Zeit.

Die Zeile 7 kann auf einfache Weise so implementiert werden, dass man die Knotenmenge in Q einmal durchgeht und sich den Knoten mit dem geringsten k–Wert dabei heraussucht. Hierfür benötigt man beim ersten Durchgang n, bei dem zweiten Durchgang $n - 1$, usw. viele Schritte, da dies genau der jeweiligen Anzahl von Knoten in Q entspricht. Wegen

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2} = O(n^2)$$

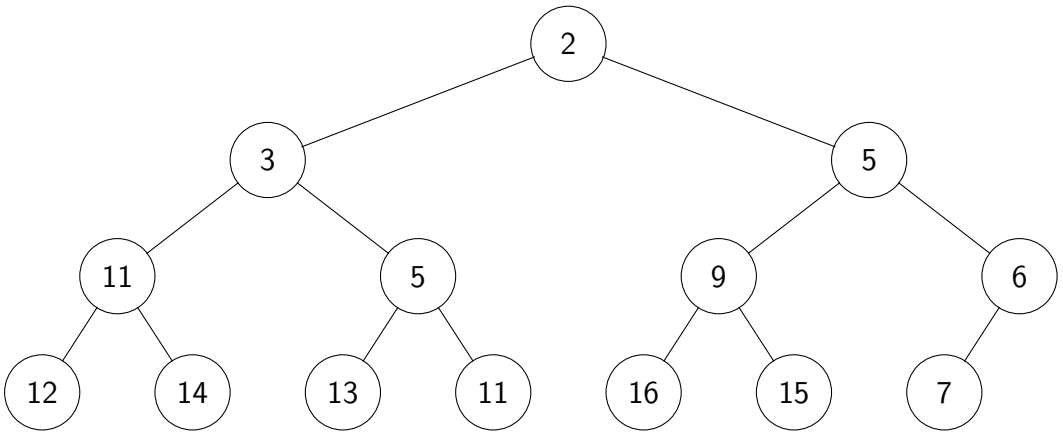
benötigen alle Ausführungen von Zeile 7 insgesamt $O(n^2)$ viel Zeit (die Summenformel für die ersten n Zahlen wird in einem Beispiel auf Seite 91 hergeleitet). Die Zeitkomplexität des gesamten Verfahrens ist also durch $O(n + |E|) + O(n^2) = O(n^2 + |E|)$ beschränkt. Ferner ist jeder der n Knoten mit höchstens jedem anderen der $n - 1$ anderen Knoten verbunden, d.h. es gibt höchstens $n(n-1) = O(n^2)$ viele Kanten insgesamt. Also lässt sich der Ausdruck $O(n^2 + |E|)$ wegen $|E| = O(n^2)$ auf $O(n^2)$ vereinfachen. □

Beachten Sie, dass normalerweise die Anzahl der Kanten in einem Graph deutlich kleiner als $O(n^2)$ ist. Bei Straßenkarten ist z.B. jeder Knoten (entspricht einer Straßenverzweigung) nur mit einigen Knoten aus seiner Umgebung verbunden. Wenn man die Anzahl der benachbarten Knoten z.B. gegen 20 abschätzen kann, so besitzt der Graph maximal $20 \cdot n = O(n)$ viele Kanten. Hier wirkt sich dann die zu einfach gewählte Implementierung der Zeile 7 negativ auf die Zeitkomplexität aus (denn das restliche Verfahren läuft in optimaler linearer Zeit). Mit dem Einsatz anderer Datenstrukturen lässt sich die Komplexität deshalb erheblich verbessern:

- Mit sogenannten *Heaps* läuft der Algorithmus von DIJKSTRA in $O((n + m) \log n)$ viel Zeit, wobei m die Kardinalität von $|E|$ bezeichnet. Im obigen Fall $m = O(n)$ beträgt die Zeitkomplexität dann also nur noch $O(n \log n)$.

- Mit sogenannten *Fibonacci–Heaps* (einer relativ komplexen Datenstruktur) lässt sich die Komplexität sogar auf $O(n \log n + m)$ drücken. Eine Laufzeit von $O(n \log n)$ lässt sich hier also sogar noch im Fall $m = O(n \log n)$ erreichen (mit Heaps würde die Laufzeit dagegen $O(n \log^2 n)$ betragen). In Abschnitt 2.6 werden wir zeigen, dass die Implementierung mit Fibonacci–Heaps sogar optimal ist.

Im Folgenden besprechen wir die Implementierung des Verfahrens von DIJKSTRA mit Heaps im Detail. Dazu müssen wir zunächst klären, was ein solcher Heap überhaupt ist. Betrachten Sie dazu den folgenden Graph:



Ein *Heap* ist zunächst ein *Binärbaum*, d.h. jeder Knoten hat prinzipiell genau zwei Nachfolger. Nur die Knoten in der vorletzten Reihe haben evtl. nur einen Nachfolger (so wie hier der Knoten mit der Zahl 6) oder sogar gar keinen. Würde beispielsweise der Knoten mit der Zahl 7 fehlen, so würde der Knoten mit der Zahl 6 ohne Nachfolger dastehen. Ohne Nachfolger sind natürlich auch die Knoten in der untersten Reihe. Die Knoten stehen in jeder Zeile immer dicht an dicht von links nach rechts aufgereiht da, d.h. eine weitere Zahl würde erst den rechten Nachfolger des Knotens mit der Zahl 6 bilden, bevor mit der nächsten Zahl eine neue Reihe eröffnet wird (die dann den linken Nachfolger der Zahl 12 bilden würde).

Zusätzlich erfüllt ein Heap (genauer gesagt ein *Min–Heap*) die sog. *Heapeigenschaft*, dass der Wert jedes Knotens immer kleiner oder gleich den Werten seiner Nachfolger ist. Beispielsweise hat hier der *Wurzelknoten* (d.h. der oberste Knoten an der Spitze des Heaps) den Wert 2, und für seine beiden Nachfolger gilt $2 \leq 3$ und $2 \leq 5$. Daraus ergibt sich direkt diese Folgerung:

2.2.4 Lemma Das Minimum m aller beteilgten Zahlenwerte ist im Wurzelknoten gespeichert.

Beweis: Falls nicht, so befindet sich ein Wert $M > m$ im Wurzelknoten, und auf dem eindeutigen Weg vom Wurzelknoten mit dem Wert M zu einem Knoten mit dem Wert m muss mindestens einmal die Heapeigenschaft verletzt sein. □

Außerdem halten wir fest:

2.2.5 Lemma Die *Höhe* h eines Heaps (d.h. die Anzahl der beteiligten Zahlenreihen, im obigen Beispiel also 4) verhält sich logarithmisch zur Anzahl n der darin gespeicherten Zahlen, d.h. es gilt $h = O(\log n)$.

Beweis: In einem Heap der Höhe h sind abgesehen von der untersten Zahlenreihe alle Reihen voll besetzt. In der obersten Reihe eines Heaps befindet sich dabei nur genau ein Knoten, nämlich der Wurzelknoten. In der zweiten Reihe sind es schon zwei, in der dritten vier, und allgemein in der k–ten Zeile 2^{k-1} Knoten. In der evtl. nur partiell besetzten letzten (h–ten) Zahlenreihe befinden sich zwischen einem und 2^{h-1} viele Knoten. Somit erfüllt n die folgende Ungleichung:

$$1 + \sum_{k=1}^{h-1} 2^{k-1} \leq n \leq 2^{h-1} + \sum_{k=1}^{h-1} 2^{k-1} = \sum_{k=1}^{h} 2^{k-1} \ ,$$

also

$$1 + \sum_{k=0}^{h-2} 2^k \leq n \leq \sum_{k=0}^{h-1} 2^k$$

und somit

$$1 + 2^{h-1} - 1 \leq n \leq 2^h - 1 \iff 2^{h-1} \leq n < 2^h$$

Insbesondere ergibt die Anwendung des Logarithmus auf beiden Seiten des vorderen Teils der Ungleichung

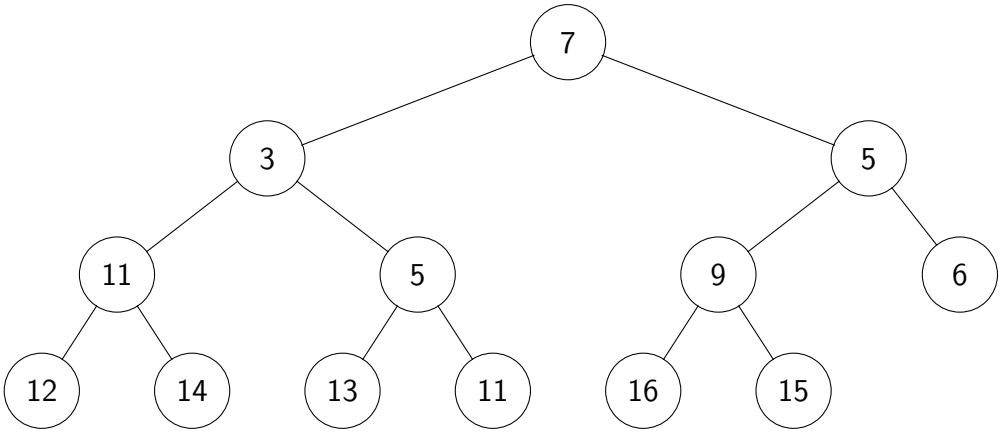
$$h = h - 1 + 1 \leq \log_2 n + 1 = O(\log n)$$

und damit die Behauptung. □

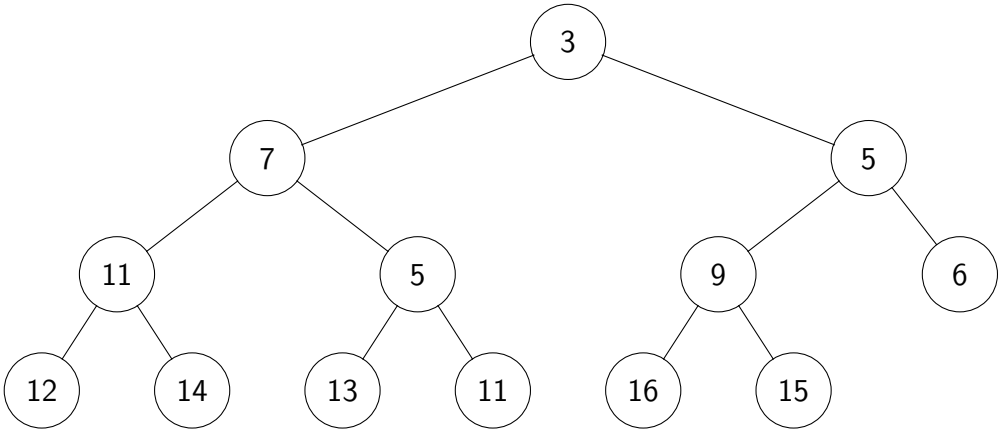
Auf einem Heap mit n Zahlen kann man diverse Operationen effizient ausführen. Wir wissen z.B. bereits, dass man die kleinste Zahl in konstanter Zeit im Wurzelknoten ablesen kann. Vergleichen Sie dies mit dem Aufwand, eine kleinste Zahl in einem unsortierten Array mit n Zahlen ausfindig zu machen: hier schlägt eine einfache Suche mittels einer einfachen `for`–Schleife über alle Zahlen mit $O(n)$ viel Aufwand zu Buche. Man kann in einem Heap das kleinste Element aber auch recht kostengünstig entfernen:

2.2.6 Lemma Die Entnahme eines kleinsten Elements aus einem Heap mit n Zahlen kostet nur $O(\log n)$ viel Zeit.

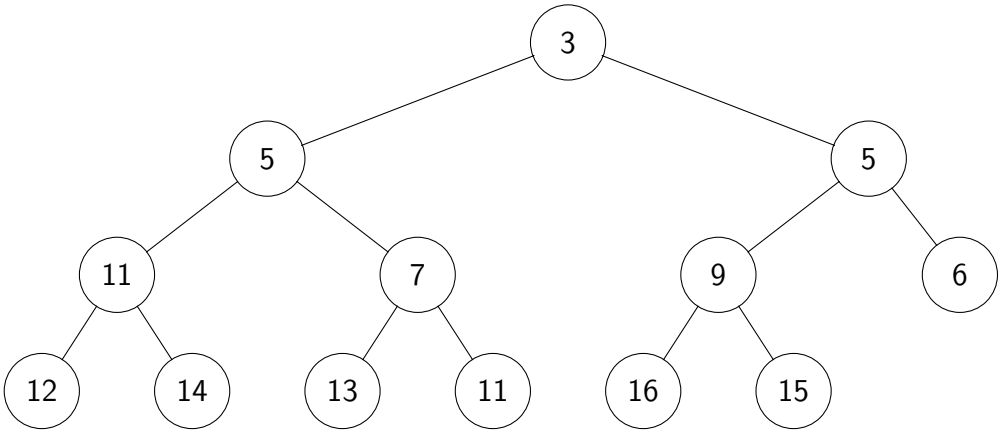
Beweis: Die folgende Technik führt zum Ziel. Zunächst entfernen wir die kleinste Zahl aus dem Wurzelknoten und verlegen dorthin die letzte Zahl aus der untersten Reihe, so dass der Baum um eine Zahl kleiner wird und dann nur noch $n - 1$ Elemente enthält. Im obigen Beispiel würde sich dieser Baum ergeben:



Nun ist ein solcher Baum im Allgemeinen kein Heap mehr. Hier ist zum Beispiel jetzt die Zahl 7 größer als die beiden Nachfolgerzahlen 3 und 5. Wir lassen die 7 daher „versickern“ und vertauschen sie mit dem kleineren der beiden Nachfolger, hier also der 3:



An der neuen Position verletzt die 7 jedoch die Heapbedingung immer noch, da der rechte Nachfolger 5 kleiner als die 7 ist. Wir versickern also weiter:



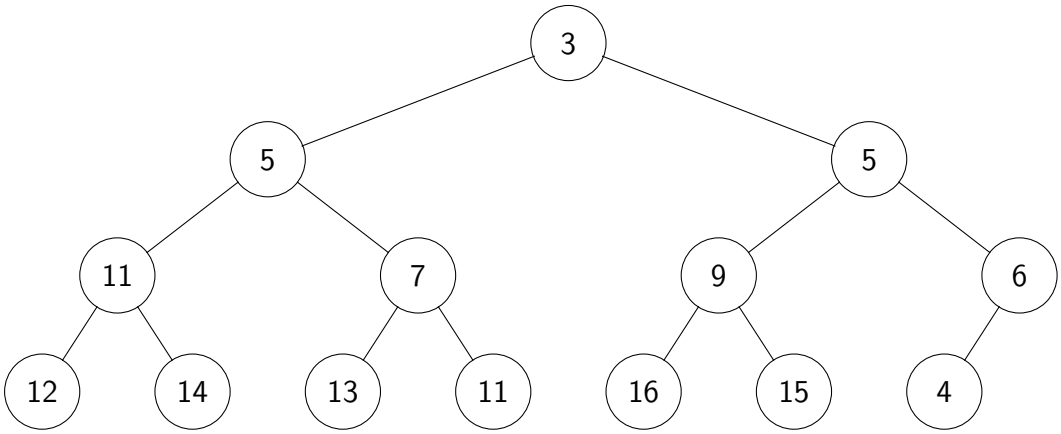
Dieser Heap ist wieder bereits vollständig korrekt. Evtl. wäre nochmals ein abschließen-

der Tausch erforderlich gewesen, spätestens dann aber hätten wir die unterste Ebene erreicht und den Versickerungsprozess abgeschlossen. Da uns jeder Tausch eine Ebene tiefer bringt und es nach Lemma 2.2.5 überhaupt nur $O(\log n)$ viele Ebenen gibt, benötigt die gesamte Operation nur $O(\log n)$ viel Zeit. □

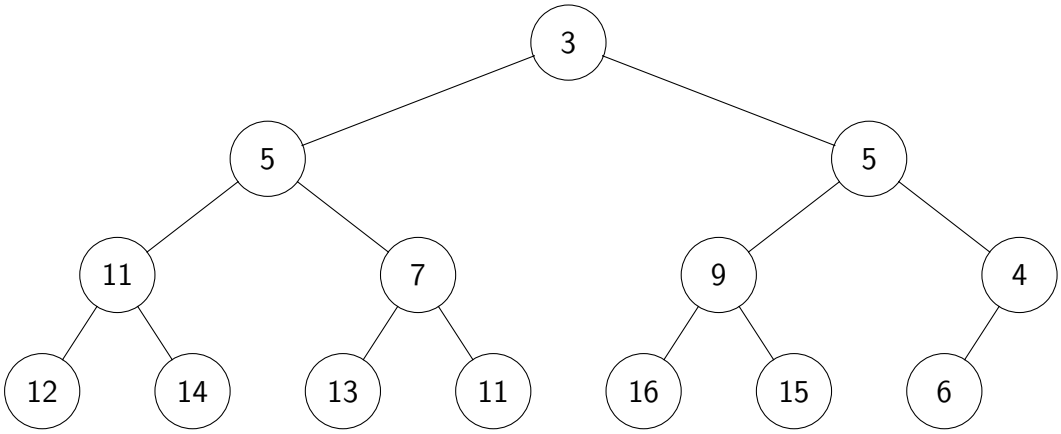
Umgekehrt lässt sich in logarithmischer Zeit auch eine neue Zahl hinzufügen:

2.2.7 Lemma Eine neue Zahl kann in $O(\log n)$ viel Zeit zu einem Heap mit n Knoten hinzugefügt werden.

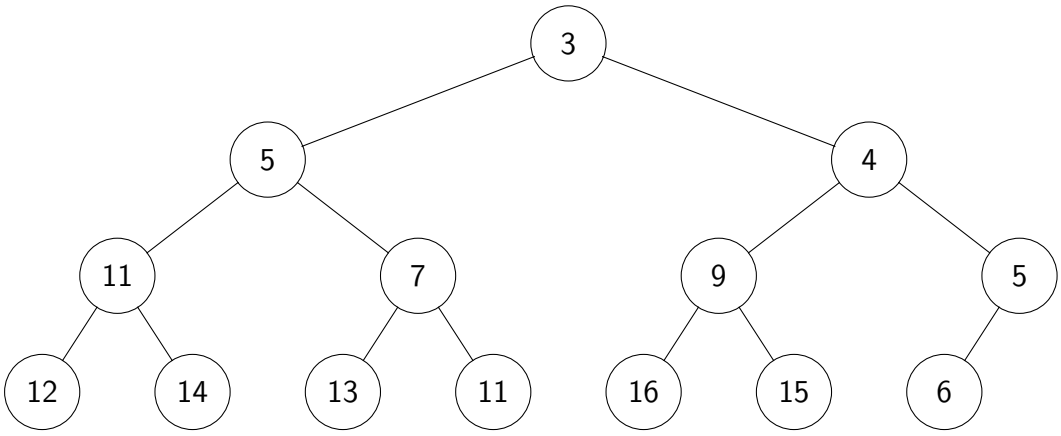
Beweis: Wir stellen die neue Zahl an das Ende der letzten Zahlenreihe bzw. (falls diese bereits komplett gefüllt ist) ganz nach links in eine neue Zahlenreihe, was die Höhe um 1 erhöht. In unserem zuletzt besprochenen Heap (nach der Löschaktion) ist die letzte Reihe aber noch nicht vollständig besetzt. Durch die Hinzunahme von z.B. der Zahl 4 würden wir also zunächst dieses Ergebnis erhalten:



Auch jetzt ist der Baum im Allgemeinen kein gültiger Heap mehr (hier ist z.B. die 4 kleiner als ihr Vorgänger 6). Deshalb verwenden wir erneut die Versickerungstechnik, diesmal aber in umgekehrter Richtung von unten nach oben. Als erstes tauschen wir die 4 und die 6 aus:



Nun verletzt die Zahl 4 immer noch die Heapeigenschaft (die 5 ist größer als die 4), also führen wir einen weiteren Tausch aus:

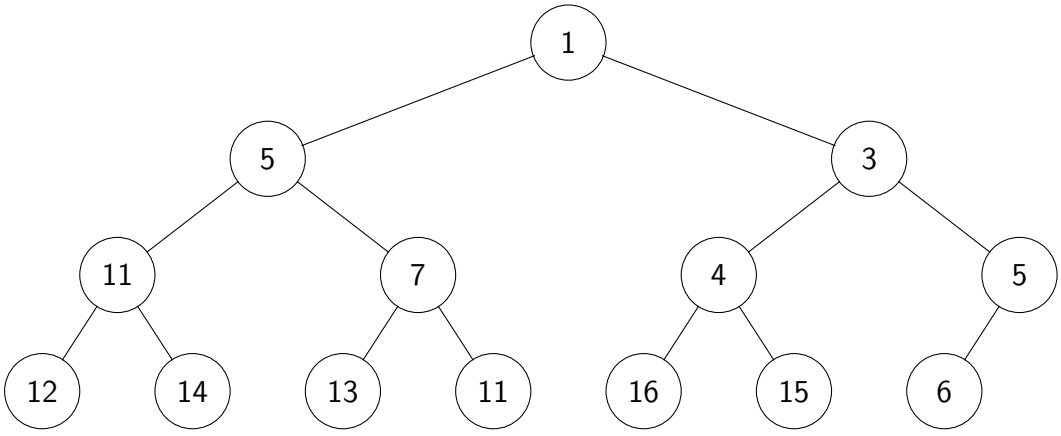


Nun liegt wieder ein korrekter Heap vor. Evtl. sind in der Praxis noch weitere Vertauschungen nötig, aber nach spätestens $O(\log n)$ vielen Schritten haben wir den Wurzelknoten erreicht, so dass der Versickerungsprozess wie behauptet in $O(\log n)$ viel Zeit endet. □

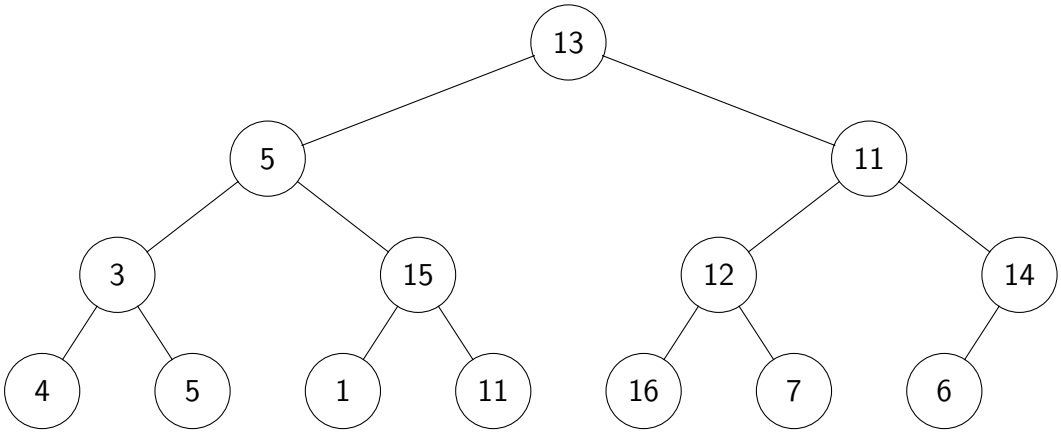
2.2.8 Lemma In einem Heap mit n Zahlen kann eine dieser Zahlen in $O(\log n)$ viel Zeit durch eine kleinere Zahl ersetzt werden.

Beweis: Das passende Verfahren für diese Operation funktioniert ganz ähnlich wie in Lemma 2.2.7. Zuerst wird die Zahl in dem entsprechenden Knoten vermindert, anschließend erfolgt eine evtl. notwendige Versickerung nach oben. Selbst wenn anfangs die Verminderung der Zahl auf unterster Ebene erfolgt, endet der Versickerungsprozess nach spätestens $O(\log n)$ vielen Schritten. □

Wird beispielsweise der Knoten mit der Zahl 9 auf 1 vermindert, so sieht der resultierende Heap wie folgt aus:

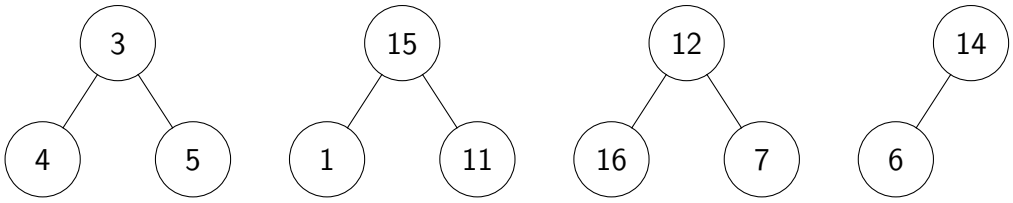


Nun stellt stellt sich noch die Frage, wie man einen Heap mit n Zahlen überhaupt aufbauen kann. Interessanterweise gelingt dies mit der folgenden Technik in Linearzeit. Betrachten Sie dazu nochmals den folgenden Binärbaum, in dem wir als Beispiel alle Zahlen aus der letzten Konfiguration ungeordnet hinein geschrieben haben:

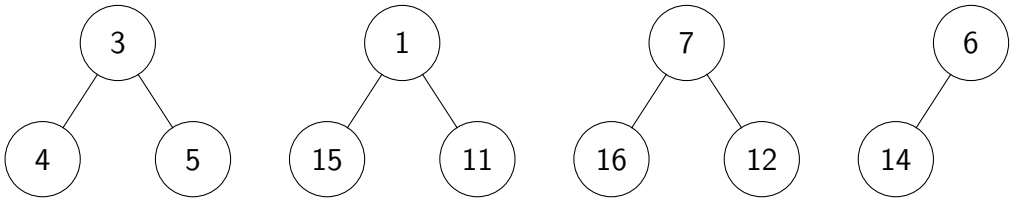


Wir gehen nun alle Reihen von unten nach oben durch und betrachten jeden Knoten in einer solchen Reihe als Wurzelknoten eines „lokalen“ Binärbaums, zu dem alle darunterhängenden Knoten gehören. Jeder solche lokale Binärbaum wird in einen Heap umgewandelt (dazu gleich mehr), danach wird mit dem nächsten Knoten fortgesetzt. Wenn wir dies schließlich in der obersten Reihe für den dort stehenden „echten“ Wurzelknoten unseres Gesamtbaums durchgeführt haben, ist der gewünschte Heap aufgebaut.

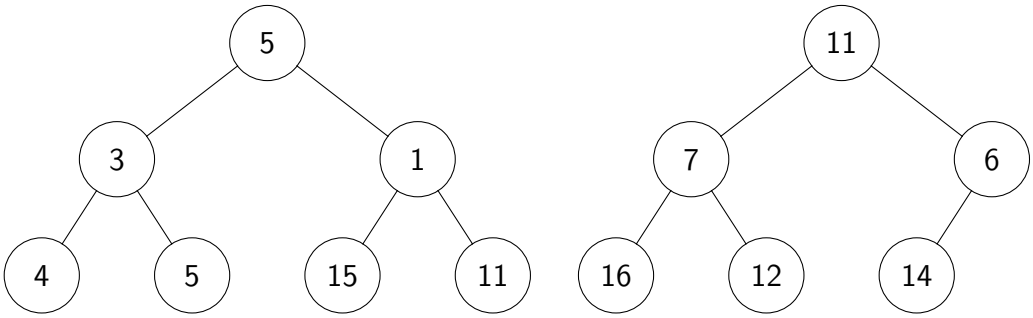
In der untersten Reihe ist nichts zu tun, denn jeder dort stehende Knoten ist frei von Nachfolgern. Somit besteht jeder lokale Baum nur aus seinem einen Wurzelknoten, welcher trivialerweise auch einen korrekten Heap bildet. Betrachten wir daher nun die vorletzte Reihe. In unserem Beispiel befinden sich dort die folgenden vier lokalen Binärbäume:



Der erste Baum links bildet bereits zufälligerweise einen Heap, die anderen drei jedoch nicht. Wir führen daher in jedem dieser Bäume Versickerungsprozesse ausgehend von den Wurzelknoten nach unten durch (so wie in Lemma 2.2.6) und erhalten:

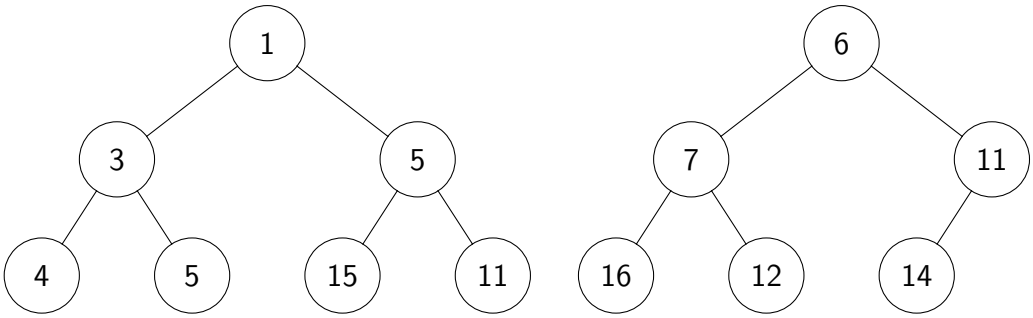


Eine Reihe höher (hier also bereits in der zweithöchsten) stellen sich die dortigen beiden Binärbäume nun wie folgt dar:

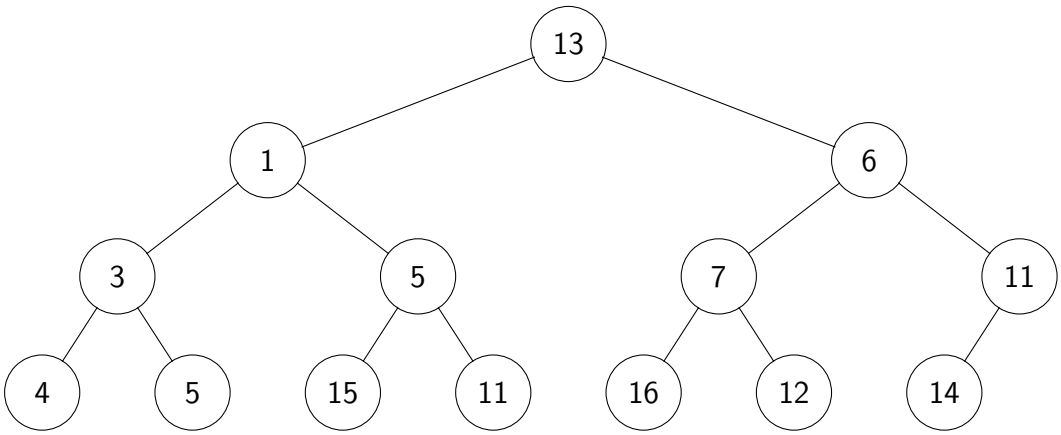


Auch hier reicht es zur Konstruktion der Heaps aus, jeweils nur den Wurzelknoten zu versickern, da sich im linken und rechten Ast jeweils schon ein fertig aufgebauter Heap befindet.

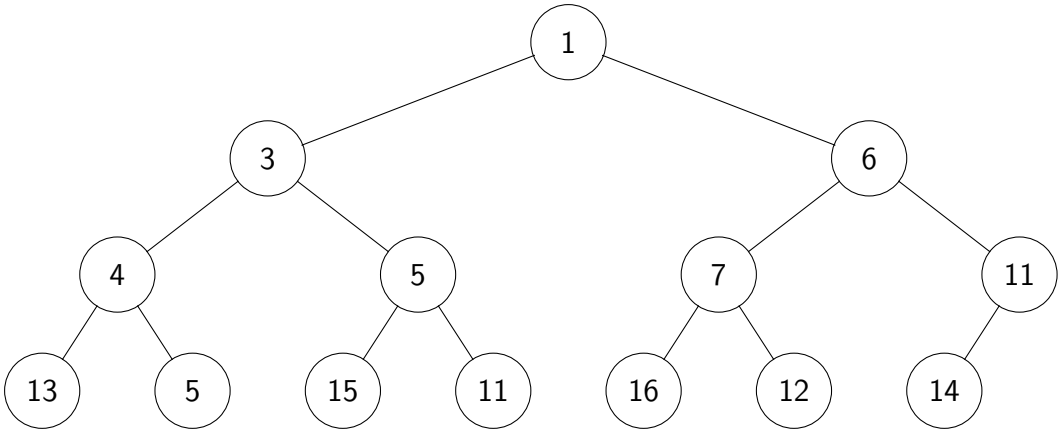
Wir erhalten demnach:



Auf oberster Ebene angelangt sieht der vollständige Binärbaum nun so aus:



Durch das Versickern des Wurzelknotens erhalten wir schließlich den gesuchten vollständigen Heap:



Es ist nicht offensichtlich, dass dieses Verfahren nur linear viel Zeit benötigt. Tatsächlich erfordert die Analyse ein wenig höhere Mathematik.

2.2.9 Lemma Der Aufbau eines Heaps mit n Knoten gelingt in $O(n)$ viel Zeit.

Beweis: Der Zeitaufwand für die beschriebene Konstruktionstechnik ist proportional zu der Anzahl der durchgeführten Tauschoperationen. Wir werden nun zeigen, dass weniger als $2n = O(n)$ solcher Tauschoperationen benötigt werden.

Wie in Lemma 2.2.5 gesehen wird die Höhe h des konstruierten Heaps die Bedingung

$$2^{h-1} \leq n < 2^h$$

erfüllen. Für alle Knoten in der untersten Reihe (davon gibt es bis zu 2^{h-1} Stück) sind überhaupt keine Tauschoperationen notwendig, da wie gesehen diese jeweils einen trivialen Heap mit nur einem Knoten bilden. Für alle 2^{h-2} Knoten in der vorletzten (d.h. $(h-1)$–sten) Reihe ist für den evtl. anfallenden Versickerungsprozeß des jeweiligen Wurzelknotens nur höchstens ein Tauschvorgang notwendig, da diese Heaps nur die Höhe 2 haben. Analog sind für die 2^{h-3} Knoten in der drittletzten (d.h. $(h-2)$–ten) Reihe nur je höchstens zwei Vertauschungen notwendig, und allgemein benötigen wir in der $(h-k)$–ten Reihe $(1 \leq k < h)$ für die Konstruktion der 2^{h-k-1} Heaps nie mehr als je k Vertauschungen. Somit lässt sich die gesamte Konstruktion mit höchstens

$$\sum_{k=1}^{h-1} k \cdot 2^{h-k-1}$$

Vertauschungsschritten realisieren. Wir zeigen nun, dass

$$\sum_{k=1}^{h-1} k \cdot 2^{h-k-1} < 2n = O(n)$$

gilt. Dazu rechnen wir

$$\sum_{k=1}^{h-1} k \cdot 2^{h-k-1} = 1 \cdot 2^{h-2} + 2 \cdot 2^{h-3} + \ldots (h-1) \cdot 2^0 = \sum_{k=0}^{h-2} (h-(k+1)) \cdot 2^k$$

und weiter

$$\sum_{k=0}^{h-2}(h-(k+1))\cdot 2^k = h\sum_{k=0}^{h-2}2^k - \sum_{k=0}^{h-2}(k+1)\cdot 2^k = h(2^{h-1}-1) - \sum_{k=0}^{h-2}(k+1)\cdot 2^k \ .$$

Definieren wir nun

$$f(x) := \sum_{k=0}^{h-2}(k+1)\cdot x^k \ ,$$

so lässt sich der letzte Term auch als

$$h\cdot(2^{h-1}-1) - f(2)$$

schreiben. Um eine geschlossene Form für $f(2)$ zu finden, integrieren wir die Funktion f und leiten sie gleich wieder ab, so dass sich nichts verändert. Wir erhalten dann

$$F(x) := \int f(x)\,dx = \int \sum_{k=0}^{h-2}(k+1)\cdot x^k\,dx = \sum_{k=0}^{h-2}\int (k+1)\cdot x^k\,dx = \sum_{k=0}^{h-2} x^{k+1} \ .$$

Wegen

$$\sum_{k=0}^{h-2} x^{k+1} = x\sum_{k=0}^{h-2} x^k = x\cdot\frac{x^{h-1}-1}{x-1} = \frac{x^h - x}{x-1}$$

ergibt sich somit für die Ableitung (Quotientenregel!):

$$F'(x) = f(x) = \frac{(hx^{h-1}-1)(x-1)-(x^h-x)}{(x-1)^2} \ .$$

Wir ersparen uns eine Vereinfachung dieses Terms, sondern werten ihn direkt an der Stelle $x=2$ aus, da wir ja nur an $f(2)$ interessiert sind:

$$f(2) = \frac{(h2^{h-1}-1)(2-1)-(2^h-2)}{(2-1)^2} = (h2^{h-1}-1)-(2^h-2) = (h-2)2^{h-1}+1$$

Setzen wir dies in die oben erhaltenen Abschätzung für die Gesamtanzahl aller Vertauschungen ein, so erhalten wir

$$h\cdot(2^{h-1}-1) - f(2) = h\cdot(2^{h-1}-1) - \Big((h-2)2^{h-1}+1\Big) = 2\cdot 2^{h-1} - h - 1 \ .$$

Nun gilt $2^{h-1} \le n$ und daher

$$2\cdot 2^{h-1} - h - 1 \le 2n - h - 1 < 2n = O(n) \ ,$$

so dass die Behauptung bewiesen ist. □

Nun könnte man meinen, dass die Verwaltung eines Heaps aufwendige Zeigerstrukturen usw. zur Verwaltung des Binärbaums erfordert. Dies ist aber nicht der Fall. Ein einfaches Array mit den Indizes $i = 1,\ldots,n$ reicht dafür aus. Man muss lediglich die Zahlen Reihe für Reihe von oben nach unten und pro Reihe von links nach rechts nacheinander in das Array eintragen. Für den fertig aufgebauten Heap von Seite 61 ergibt sich z.B. dieser Inhalt:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	1	3	6	4	5	7	11	13	5	15	11	16	12	14

Nun gilt (auf Seite 61 lassen sich die Beispiele leicht nachvollziehen):

- Die beiden Nachfolger eines Knotens mit Index i stehen an den Positionen $2i$ und $2i+1$. Beachten Sie, dass für die unteren (hier also hinteren) Knoten evtl. der rechte oder gar beide Nachfolger fehlen. Dies kann man leicht daran erkennen, dass der berechnete Index über den maximal erlaubten Wert n hinausragt. Beispielsweise hat hier der Knoten an der Position $i = 7$ (also die 11) nur einen linken Nachfolger (an der Position $2 \cdot 7 = 14$), aber keinen rechten.
- Umgekehrt kann man für jeden Knoten mit Index i den zugehörigen Vorgängerknoten ermitteln, indem man i einfach mit einer Ganzzahl–Division durch 2 (also abgerundet ohne Nachkommastellen) teilt. Beispielweise hat der Knoten an der Position $i = 13$ (also die 12) seinen Vorgänger an der Position $13/2 = 6$. Nur der Knoten an der Position $i = 1$ hat keinen Vorgänger — es ist der Wurzelknoten.

Wenn man nun beispielsweise das Minimum aus dem Heap entfernt — es handelt sich immer um den Knoten an der Position 1 — so würde man gemäß des Verfahrens aus Lemma 2.2.6

- zuerst die letzte Zahl 14 nach vorne an die erste Position 1 stellen:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	14	3	6	4	5	7	11	13	5	15	11	16	12	–

- dann die 14 an der Position 1 mit den Zahlen an den Positionen 2 und 3 vergleichen und versickern lassen, hier also die 14 mit der 3 tauschen:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	3	14	6	4	5	7	11	13	5	15	11	16	12	–

- dann die 14 an der Position 2 mit den Zahlen an den Positionen 4 und 5 vergleichen und versickern lassen, hier also die 14 mit der 4 tauschen:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	3	4	6	14	5	7	11	13	5	15	11	16	12	–

- dann die 14 an der Position 4 mit den Zahlen an den Positionen 8 und 9 vergleichen und versickern lassen, hier also die 14 mit der 5 tauschen:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	3	4	6	5	5	7	11	13	14	15	11	16	12	–

- und schließlich die Operation beenden, denn weitere Zahlen an den Positionen 16 und 17 gibt es nicht.

Beachten Sie, dass nun der Heap nur noch die Positionen 1 bis 13 belegt. An der freigewordenen Position 14 können wir die entnommene kleinste Zahl 1 speichern:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	3	4	6	5	5	7	11	13	14	15	11	16	12	1

Nun können wir nochmals die jetzt kleinste Zahl 3 entnehmen, den Heap entsprechend aktualisieren und anschließend die 3 an der freigewordenen Position 13 speichern:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	4	5	6	12	5	7	11	13	14	15	11	16	3	1

Wenn wir dies solange wiederholen, bis der Heap komplett abgebaut ist, erhalten wir am Ende

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Inhalt	16	15	14	13	12	11	11	7	6	5	5	4	3	1

und haben somit den Inhalt des Arrays absteigend sortiert — dies ist der *Heapsort*–Algorithmus. Hätten wir statt eines Min–Heaps einen sog. *Max–Heap* verwendet (hier ist ein Knoten immer größer oder gleich den Werten seiner Nachfolger), so wäre das Array aufsteigend sortiert worden. Der Aufwand von Heapsort setzt sich somit aus dem einmaligen Aufbau des Heaps und der n–maligen Entnahme eines minimalen Elements zusammmen, d.h. die Laufzeit beträgt nach den Lemmata 2.2.9 und 2.2.6 insgesamt nur

$$O(n) + n \cdot O(\log n) = O(n \log n) \ .$$

Diese Laufzeit wird immer erreicht (im Gegensatz zu *Quicksort*, welches im „schlimmsten“ Fall immer quadratische Laufzeit verlangt) und kommt auch ohne irgendwelchen Zusatzspeicher aus (ebenfalls im Gegensatz zu Quicksort oder auch zu der gängigen Implementierung von *Mergesort*).

Zurück zu unserem Dijkstra–Algorithmus:

2.2.10 Satz Der Algorithmus von Dijkstra benötigt bei einer Implementierung mit Heaps nur $O((n+m)\log n)$ viel Zeit.

Beweis: Wir verwalten die Knotenmenge Q mit ihren aktuellen k–Werten nun in einem Min–Heap, d.h. wir speichern in jedem Knoten des Heaps ein Paar $(v, k[v])$, wobei sich die Heap–Eigenschaft natürlich an dem k–Wert orientiert. Zur Bestätigung der besseren Laufzeit müssen wir nun lediglich die Arbeitsweise des Verfahrens mit unseren Erkenntnissen über Heaps kombinieren:

- Anfangs sind alle n Knoten in Q und damit dem Heap enthalten. Der Aufbau des Heaps kann nach Lemma 2.2.9 in $O(n)$ viel Zeit bewerkstelligt werden, ist hier aber sogar besonders einfach. Denn anfangs hat nur der Startknoten s den k–Wert 0, alle anderen haben den Wert ∞. Den entsprechenden Heap mit dem Paar $(s, 0)$ im Wurzelknoten sowie allen anderen Knoten in beliebiger Reihenfolge darunter kann man natürlich auch direkt abspeichern.

- In den Zeilen 7 und 8 (siehe Seite 49) wird insgesamt n-mal ein Knoten mit dem kleinsten k–Wert aus dem Heap extrahiert. Dies kostet gemäß Lemma 2.2.6 insgesamt $n \cdot O(\log n) = O(n \log n)$ viel Zeit.
- In den Zeilen 9 bis 16 werden für jeden extrahierten Knoten seine ausgehenden Kanten auf Verbesserungen überprüft. Insgesamt wird also jede Kante nur einmal betrachtet, und im schlimmsten Fall führt jede dieser m Überprüfungen tatsächlich zu einer Verbesserung. Diese besteht im Wesentlichen darin, den k–Wert des betroffenen Knotens zu reduzieren. Dies kostet nach Lemma 2.2.8 nochmals $m \cdot O(\log n) = O(m \log n)$ viel Zeit.

Zusammen wird also nur

$$O(n) + O(n \log n) + O(m \log n) = O((n + m) \log n)$$

viel Zeit benötigt. □

Mit Hilfe der bereits auf Seite 55 erwähnten *Fibonacci–Heaps* gelingt die „Knoten–Verminderungs–Operation“ von Lemma 2.2.8 sogar in durchschnittlich nur konstanter Zeit, so dass insgesamt $O(m)$ viel Zeit für alle Wegeverbesserungen ausreicht. Daraus resultiert bei Fibonacci–Heaps die nochmalige Laufzeitverbesserung auf nur noch $O(n \log n + m)$. Wir werden aus zeitlichen Gründen diese Implementierungen jedoch nicht weiter besprechen. Interessierte Hörer seien auf die vorzügliche Darstellung in [1] verwiesen.

2.3. Der Algorithmus von Moore–Bellman–Ford

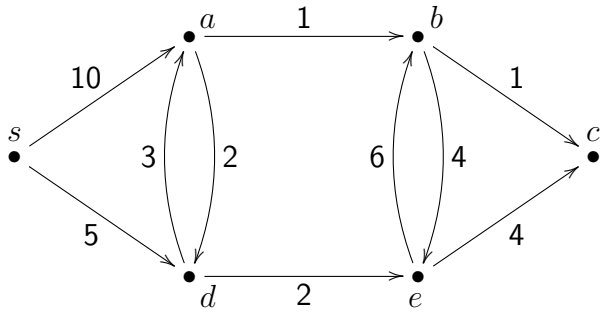
Ein Schwachpunkt des Algorithmus von Dijkstra ist die Notwendigkeit nicht–negativer Kantengewichte. Diese Einschränkung trifft auf das folgende Verfahren von Moore[1], Bellman[2] und Ford[3] nicht zu. Es muss lediglich sichergestellt werden, dass kein Kreis mit negativer Gesamtlänge existiert.

Wie schon im Algorithmus von Dijkstra werden die kürzesten Pfade zwischen einem Startknoten s und allen von s aus erreichbaren Knoten gesucht, und auch die Bedeutungen der berechneten k– und p–Attribute sind identisch. Der Algorithmus von Moore–Bellman–Ford sieht zudem dem Algorithmus von Dijkstra recht ähnlich.

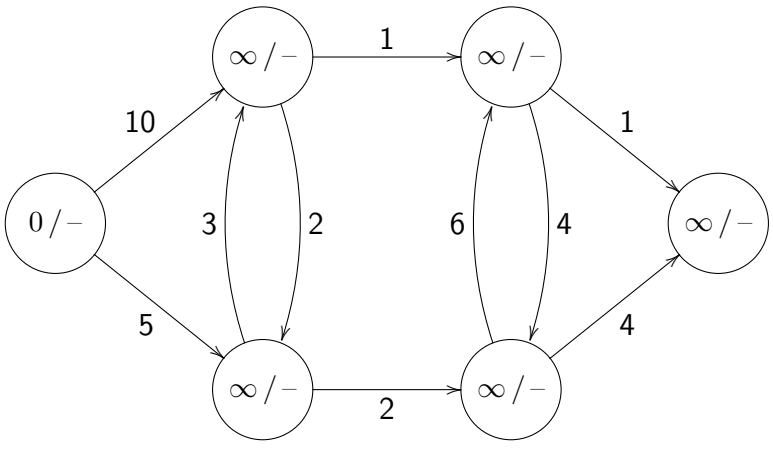
2.3.1 Beispiel Wir wenden auch diesen Algorithmus auf den Graph aus dem Beispiel 2.1.2 an:

[1] Edward F. Moore, *1925 Baltimore, †2003 Madison, US–amerikanischer Informatiker, 1966–1985 Professor für Mathematik und Infomatik an der University of Wisconsin.
[2] Richard Bellman, *1920, †1984, US–amerikanischer Mathematiker, ab 1965 Professor für Mathematik an der University of Southern California.
[3] Lester R. Ford, *1927, US–amerikanischer Mathematiker.

```
Algorithmus von Moore-Bellman-Ford

 1    for v ∈ V \ {s} do {
 2        k[v] := ∞;
 3    }
 4    k[s] := 0;
 5    for i := 1 to n − 1 do { // n = Anzahl der Knoten
 6        for (v, w) ∈ E do {
 7            if k[v] + c((v, w)) < k[w] then {
 8                k[w] := k[v] + c((v, w));
 9                p[w] := v;
10            }
11        }
12    }
```



Wir notieren wieder für jeden Knoten $v \in V$ die aktuelle Distanz $k[v]$ sowie den aktuellen Vorgängerknoten $p[v]$ in dem zugehörigen Kreis des Knotens als Paar $(k[v]/p[v])$. Die ersten vier Programmzeilen sind mit dem Algorithmus von Dijkstra identisch:



Die äußere `for`–Schleife wird nun wegen $n = |V| = 6$ für die Werte $i = 1, 2, \ldots, 5$ ausgeführt. Bei jedem dieser Durchläufe wird jede Kante von der inneren `for`–Schleife einmal ausgewählt und wie in dem Algorithmus von DIJKSTRA auf eine mögliche Pfadverbesserung hin überprüft (Zeilen 7–10). Es ist dabei unwesentlich, in welcher Reihenfolge die Kanten von der inneren `for`–Schleife durchlaufen werden (die Reihenfolge kann sogar jedesmal anders gewählt werden). Um das konkrete Beispiel fortzuführen, legen wir aber jetzt eine bestimmte Reihenfolge fest, z.B.

$$(a,b),(a,d),(b,c),(b,e),(d,a),(d,e),(e,b),(e,c),(s,a),(s,d) \ .$$

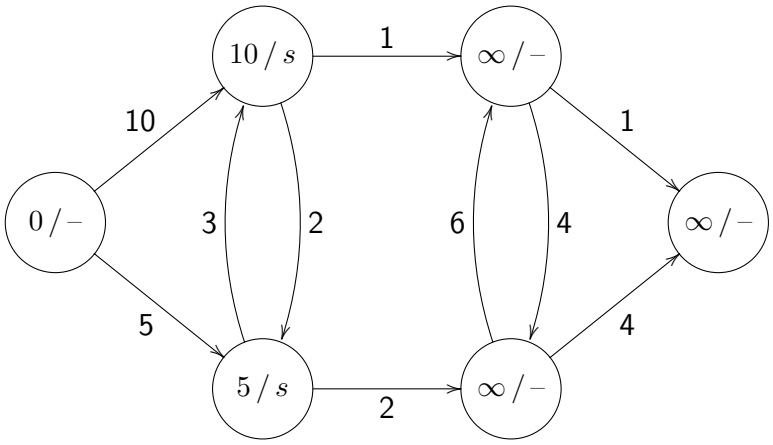
Bei der ersten Verarbeitung der inneren `for`–Schleife (also $i = 1$) wirken sich dann nur die beiden letzten von s ausgehenden Kanten aus und verbessern den Wert von $k[a]$ auf

$$k[s] + c((s,a)) = 0 + 10 = 10$$

sowie den Wert von $k[d]$ auf

$$k[s] + c((s,d)) = 0 + 5 = 5 \ .$$

Also gilt bis jetzt:



Bei dem nächsten Ausführung der kompletten inneren `for`–Schleife (in der Hauptschleife gilt jetzt $i = 2$) führt zuerst die Kante (a,b) wegen

$$k[a] + c((a,b)) = 10 + 1 = 11 < k[b] = \infty$$

zu einer Verbesserung von $k[b]$ auf den Wert 11. Die nächste Kante (a,d) bleibt dagegen wegen

$$k[a] + c((a,d)) = 10 + 2 = 12 \geq k[d] = 5$$

wirkungslos. Da jetzt $k[b]$ einen endlichen Wert besitzt, führen die beiden nächsten Kanten (b,c) und (b,e) auch zu Verbesserungen bei den Entfernungen $k[c]$ und $k[e]$, nämlich

$$k[b] + c((b,c)) = 11 + 1 = 12 \quad \text{bzw.} \quad k[b] + c((b,e)) = 11 + 4 = 15 \ .$$

Die Betrachtung der nächsten Kanten (d, a) und (d, e) verkürzt wegen

$$k[d] + c((d, a)) = 5 + 3 = 8 < k[a] = 10$$

und

$$k[d] + c((d, e)) = 5 + 2 = 7 < k[e] = 15$$

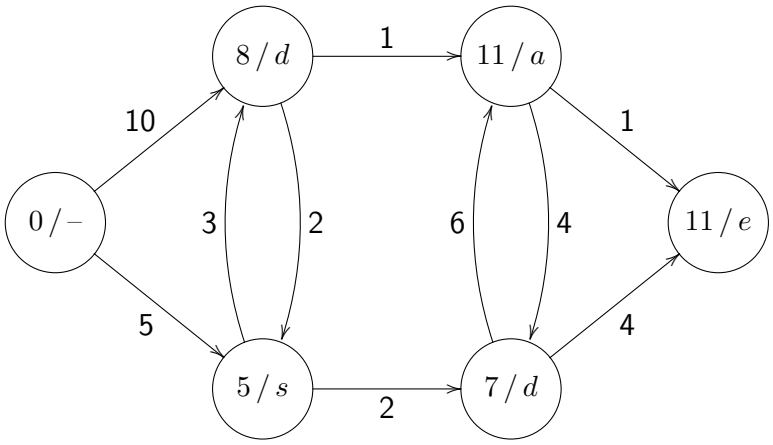
die Wege zu den Knoten a und e, während die beiden letzten Kanten (e, b) und (e, c) wegen

$$k[e] + c((e, b)) = 7 + 6 = 13 \geq k[b] = 11$$

und

$$k[e] + c((e, c)) = 7 + 4 = 11 < k[c] = 12$$

nur bei c eine Änderung hervorrufen. Also gilt nun insgesamt:



Die nächste Untersuchung aller Kanten (in der Hauptschleife gilt jetzt $i = 3$) verändert nochmals einige k– und p–Werte. So führt die erste Kante (a, b) wegen

$$k[a] + c((a, b)) = 8 + 1 = 9 < k[b] = 11$$

zu einer Verbesserung von $k[b]$ auf den Wert 9. Die nächste Kante (a, d) wirkt sich wegen

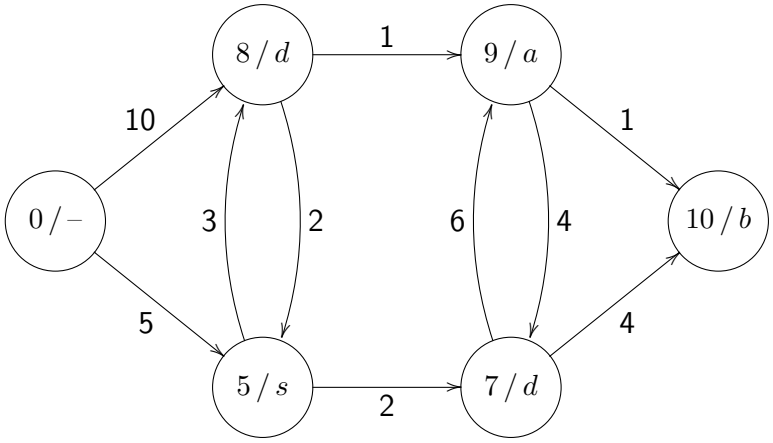
$$k[a] + c((a, d)) = 8 + 2 = 10 \geq k[d] = 5$$

nicht aus, wohl aber die darauf folgende Kante (b, c). Durch sie verbessert sich wegen

$$k[b] + c((b, c)) = 9 + 1 = 10 < k[c] = 11$$

der Wert $k[c]$ auf den Wert 10.

Die restlichen Kanten $(b, e), (d, a), (d, e), (e, b), (e, c), (s, a), (s, b)$ führen zu keiner weiteren Änderung mehr. Somit gilt nun insgesamt:



Ein Vergleich mit Beispiel 2.2.1 zeigt, dass bereits jetzt das (korrekte) Endergebnis vorliegt. Die zwei restlichen Ausführungen der äußeren `for`–Schleife (also für $i = 4$ und $i = 5$) bleiben deshalb ohne Wirkung.

2.3.2 Satz Der Algorithmus von Moore–Bellman–Ford arbeitet korrekt. Seine Laufzeit beträgt $O(nm)$ mit $n = |V|$ und $m = |E|$.

Beweis: Die behauptete Laufzeit $O(nm)$ ist klar, da die äußere Schleife $(n-1)$–mal und die innere Schleife jeweils m–mal durchlaufen wird. Für die Korrektheit zeigen wir durch vollständige Induktion, dass vor dem $(i+1)$–sten (bzw. nach dem i–ten) Durchlauf der äußeren `for`–Schleife immer gilt:

- Wenn ein Knoten $v \in V$ von s aus über einen kürzesten Pfad mit höchstens i Kanten erreichbar ist, so enthält $k[v]$ die entsprechenden Kosten, und die Vorgängerverweise sind korrekt eingetragen.
- Für alle von s aus unerreichbaren Knoten v gilt $k[v] = \infty$.

Beachten Sie, dass die Invariante über die restlichen Knoten nichts aussagt. Hierbei handelt es sich um alle Knoten v, die zwar von s aus erreichbar sind, deren kürzeste Pfade aber über mehr als i Kanten verlaufen. Hier können dann sowohl $k[v]$ als auch $p[v]$ temporär inkorrekte Werte enthalten.

$\underline{i = 0}$: Die Pfade dürfen keine einzige Kante besitzen. Also ist jeder Knoten $v \neq s$ von s aus unerreichbar. Offenbar wird daher für alle $v \in V$ der Wert $k[v]$ in den ersten vier Programmzeilen korrekt initialisiert. Dies ist aber gerade der Zeitpunkt, wenn der Algorithmus vor der $(i+1)$–sten (also ersten) Ausführung der äußeren `for`–Schleife steht.

$\underline{i \to i+1}$: Ein Pfad von s zu einem beliebigen anderen Knoten $w \in V$ darf nun die Länge $i+1$ haben. Ein solcher Pfad verläuft von s über den Vorgängerknoten v von w bis zum Endknoten w. Der Abschnitt von s nach v besteht dann nur aus höchstens i Kanten, und die dafür geringsten Kosten sind gemäß Induktionsannahme in dem Attribut $k[v]$ verfügbar. Falls der Pfad von s nach v zusammen mit den Kosten $c((v,w))$ für die abschließende Kante günstiger als der bisherige Wert $k[w]$ ist, so werden während

des $(i+1)$–sten Durchlaufs der äußeren `for`–Schleife $k[w]$ und $p[w]$ in den Zeilen 7–10 entsprechend modifiziert. Also enthält $k[w]$ anschließend den korrekten Wert.

Die Invariante ist insbesondere nach dem letzten $(n-1)$–ten Durchlauf gültig. Ein kürzester Pfad von s zu einem beliebigen Knoten $v \in V$ besteht aber immer nur aus höchstens $n-1$ Kanten, denn andernfalls müsste zumindest ein Knoten doppelt auf dem Pfad vorkommen, und der dazwischenliegende Kreis könnte ohne Gesamtkostenerhöhung herausgeschnitten werden, da es nach Voraussetzung keine negativen Kreise gibt. Dies zeigt, dass alle k–Attribute nach den $n-1$ Iterationen der äußeren `for`–Schleife den jeweils optimalen Distanzwert repräsentieren und auch die Vorgängerknoten durch die p–Attribute korrekt ausgewiesen werden. □

Im Vergleich der Algorithmen von Dijkstra und Moore–Bellman–Ford ergeben sich folgende Unterschiede:

- Der Algorithmus von Dijkstra ist schneller. Schon im günstigsten Fall $m = O(n)$ (wie immer sei $n = |V|$ die Anzahl der Knoten und $m = |E|$ die Anzahl der Kanten) besitzt der Algorithmus von Moore–Bellman–Ford nämlich bereits eine Zeitkomplexität von $O(nm) = O(n^2)$, die der maximalen Laufzeit des Algorithmus von Dijkstra entspricht. Im Fall $m = O(n^2)$ wächst die Zeitkomplexität des Algorithmus von Moore–Bellman–Ford sogar auf $O(n^3)$ an.
- Dafür ist der Algorithmus von Dijkstra nicht so universell einsetzbar, da er nur bei nicht–negativen Kantengewichten in Frage kommt.

Bei nicht–negativen Kantengewichten wird man also den Algorithmus von Dijkstra bevorzugen und ansonsten auf den Algorithmus von Moore–Bellman–Ford ausweichen.

2.4. Ermittlung von negativen Kreisen

Wir haben im ersten Abschnitt dieses Kapitels betont, dass die schnelle Berechnung von kürzesten Pfaden die Nichtexistenz von negativen Kreisen voraussetzt. Wir wollen jetzt noch abschließend klären, wie man diese Bedingung überhaupt überprüfen kann. Überraschenderweise braucht man dazu lediglich den Algorithmus von Moore–Bellman–Ford mit einem abschließenden Test zu ergänzen, bei dem alle Kanten noch einmal durchlaufen werden. Man kann also „rückwirkend“ sicherstellen, dass die Verwendung des Algorithmus von Moore–Bellman–Ford wirklich gerechtfertigt war.

Der Programmcode des Tests auf negative Zyklen ist auf der nächsten Seite abgebildet.

2.4.1 Satz Der Test auf negative Zyklen prüft, ob irgendwelche negativen Kreise von s aus erreichbar sind (und somit Einfluss auf den Algorithmus von Moore–Bellman–Ford nehmen). Seine Laufzeit beträgt $O(nm)$ mit $n = |V|$ und $m = |E|$.

```
Algorithmus Test auf negative Zyklen

1    Führe den Algorithmus von Moore-Bellman-Ford
2    mit dem gewünschten Startknoten s ∈ V aus
3    for (v,w) ∈ E do {
4       if k[v] + c((v,w)) < k[w] then {
5          output(„G enthält einen Kreis mit negativem Gewicht.“);
6          exit;
7       }
8    }
9    output(„G ist frei von negativen Kreisen.“);
```

Beweis: Die Laufzeit setzt sich aus $O(nm)$ viel Zeit für den Algorithmus von Moore–Bellman–Ford sowie $O(m)$ viel Zeit für die anschließende `for`-Schleife zusammen, d.h. insgesamt wird $O(mn) + O(m) = O(nm)$ viel Zeit benötigt.

Falls eine Kante $(v, w) \in E$ die Bedingung $k[v] + c((v, w)) < k[w]$ erfüllt, so enthält $k[w]$ nicht die korrekten minimalen Kosten für einen Pfad von s nach w, denn ein Pfad von s nach v mit einer anschließenden Kante von v nach w führt insgesamt zu einem kürzeren Pfad. Also hat der Algorithmus von Moore–Bellman–Ford nicht korrekt gearbeitet. Dies kann aber nur geschehen, falls die Voraussetzungen für die Anwendung dieses Algorithmus nicht erfüllt waren. Demnach muss G einen Kreis mit negativem Gewicht enthalten. Wenn der Algorithmus also die Existenz eines negativen Kreises behauptet, so ist dies in der Tat richtig. Man beachte jedoch, dass negative Kreise mit ausschließlich von s aus unerreichbaren Knoten nicht entdeckt werden, da dort alle k–Werte auf dem Wert ∞ verharren und somit die Bedingung $k[v]+c((v, w)) < k[w]$ wegen $\infty + c((v, w)) = \infty$ nicht erfüllt wird.

Angenommen, der obige Algorithmus terminiert nun mit der Aussage, dass kein solcher Kreis existiert. Dann ist die Bedingung in Zeile 4 für keine Kante erfüllt, d.h. es gilt

$$\forall (v, w) \in E\colon\ k[v] + c((v, w)) \geq k[w] \ .$$

Wir werden nun zeigen, dass dann tatsächlich jeder Kreis ein nicht–negatives Gesamtgewicht besitzt. Sei also

$$v_0 \ \to\ v_1 \ \to\ \ldots \ \to\ v_r$$

ein beliebiger Kreis in G, d.h. wir haben

$$(v_0, v_1), (v_1, v_2), \ldots, (v_{r-1}, v_r) \in E \quad \text{sowie} \quad v_r = v_0 \ .$$

Wegen den unerfüllten Bedingungen in Zeile 4 gilt nun zunächst

$$\sum_{i=1}^{r} k[v_{i-1}] + \sum_{i=1}^{r} c((v_{i-1}, v_i)) = \sum_{i=1}^{r} \Big(k[v_{i-1}] + c((v_{i-1}, v_i)) \Big) \geq \sum_{i=1}^{r} k[v_i] \ ,$$

und wegen $v_0 = v_r$ gilt zudem

$$\begin{aligned} & \sum_{i=1}^{r} k[v_{i-1}] \\ = \; & k[v_0] + k[v_1] + k[v_2] + \cdots + k[v_{r-1}] \\ = \; & k[v_1] + k[v_2] + \cdots + k[v_{r-1}] + k[v_r] \\ = \; & \sum_{i=1}^{r} k[v_i] \ . \end{aligned}$$

Ziehen wir diese Gleichung

$$\sum_{i=1}^{r} k[v_{i-1}] = \sum_{i=1}^{r} k[v_i]$$

von der vorherigen Ungleichung

$$\sum_{i=1}^{r} k[v_{i-1}] + \sum_{i=1}^{r} c((v_{i-1}, v_i)) \geq \sum_{i=1}^{r} k[v_i]$$

ab, so erhalten wir

$$\sum_{i=1}^{r} c((v_{i-1}, v_i)) \geq 0 \ ,$$

d.h. der Kreis hat wie behauptet ein positives Gesamtgewicht. □

2.5. Die A*–Strategie zur Verbesserung der Suchrichtung

In vielen Anwendungsfällen ist man gar nicht an der Ermittlung von kürzesten Wegen von einem Startknoten $s \in V$ zu *allen* anderen Knoten interessiert, sondern nur zu einem bereits bekannten Zielknoten $t \in V$. Dies ergibt dann das bekannte Start–Ziel–Problem, welches man tagtäglich z.B. an den Navigationsgeräten eingibt.

Nun zeigt sich, dass sich die von dem Dijkstra–Algorithmus untersuchten Knoten sternförmig (gemessen an der Entfernung) vom Startknoten weg ausbreiten. Wenn beispielsweise ein kürzester Weg von Aachen nach Berlin auf einer üblichen Europakarte ermittelt werden soll, dann untersucht der Dijkstra–Algorithmus auch z.B. Knoten in Richtung Paris, obwohl diese ganz offensichtlich nicht auf einem kürzesten Weg nach Berlin liegen können. Dies liegt ganz einfach daran, dass der Dijkstra–Algorithmus (im Gegensatz zu uns) nicht weiß, in welche Richtung man Berlin von Aachen aus suchen muss. Dies lässt sich jedoch in manchen Fällen verbessern.

Die zentrale Idee ist, dem Verfahren als Hilfsmittel eine Schätzfunktion $h : V \longrightarrow \mathbb{R}_0^+$ zur Verfügung zu stellen, die für jede Kante $(v, w) \in E$ die Bedingung $h(v) - h(w) \leq c((v, w))$

erfüllt. Mit einer solchen Funktion lassen sich untere Schranken für die Entfernung zweier Knoten v und w angeben. Ist nämlich

$$v = v_0 \;\rightarrow\; v_1 \;\rightarrow\; v_2 \;\rightarrow\; \ldots \;\rightarrow\; v_k = w$$

ein solcher Pfad von v nach w, so ergibt der Term $h(v) - h(w)$ wegen

$$\begin{aligned}
& h(v) - h(w) \\
= \;& h(v_0) - h(v_k) \\
= \;& h(v_0) - h(v_1) + h(v_1) - h(v_2) + h(v_2) - h(v_3) + \cdots + h(v_{k-1}) + h(v_k) \\
= \;& (h(v_0) - h(v_1)) + (h(v_1) - h(v_2)) + (h(v_2) - h(v_3)) + \cdots + (h(v_{k-1}) + h(v_k)) \\
\leq \;& c((v_0, v_1)) + c((v_1, v_2)) + c((v_2, v_3)) + \cdots + c((v_{k-1}, v_k))
\end{aligned}$$

eine untere Abschätzung für die Summe der Kosten aller benutzten Kanten, also den Gesamtkosten des benutzten Pfades. Dies gilt für jeden Weg von v nach w, insbesondere auch für die kürzesten Wege. Wenn wir nun noch $h(t) = 0$ vereinbaren, so lässt sich $h(v) = h(v) - h(t)$ für jeden Knoten v als eine Art Minimalaufwand interpretieren, der von v aus bis zum Zielknoten t noch aufgewendet werden muss.

Mit Hilfe einer solchen Schätzfunktion h modifizieren wir das ursprüngliche Dijkstra–Verfahren nun wie folgt:

- In jeder Iteration der Hauptschleife suchen wir nicht mehr nach einem Knoten v mit einem minimalen Wert $k[v]$, sondern nach einem Knoten mit einem minimalen Wert $k[v] + h(v)$. Neben den bisherigen Wegekosten werden also auch die noch zu erwartenen restlichen Kosten bei der Knotenauswahl berücksichtigt.
- Sinnvollerweise brechen wir das Verfahren ab, sobald der kürzeste Weg zum Zielknoten t bekannt ist, d.h. wenn der Knoten t aus der Menge Q ausgewählt wird. An den restlichen Knoten sind wir ja nicht mehr interessiert.

Der so entstehende Algorithmus ist auch als A^*–Strategie bekannt:

Die A^*–Strategie arbeitet nach wie vor korrekt:

2.5.1 Satz Der Algorithmus A^*–Strategie ermittelt einen kürzesten Weg von s nach t mit den richtigen k– und p–Werten (sofern natürlich t von s aus überhaupt erreichbar ist).

Beweis: Wie beim Dijkstra–Verfahren zeigen wir durch Induktion über die Anzahl i der Schleifendurchläufe der `while`–Schleife, dass beim Erreichen von Zeile 6 immer gilt:

a) Für alle $v \in Q$ enthält $k[v]$ die Kosten eines kürzesten Pfads von s nach v, wobei (von v abgesehen) nur Knoten aus $V \setminus Q$ benutzt werden dürfen. Für unter diesen Umständen unerreichbare Knoten gilt $k[v] = \infty$. Im Fall $v \neq s$ und $k[v] < \infty$ enthält $p[v]$ den Vorgängerknoten von v auf einem solchen Pfad.

b) Für alle $v \in V \setminus Q$ enthält $k[v]$ die Kosten eines kürzesten Pfads von s nach v ohne irgendwelche Restriktionen, d.h. die ermittelten Kosten sind bereits optimal. Auch

```
ALGORITHMUS A∗-STRATEGIE

 1    for v ∈ V \ {s} do {
 2       k[v] := ∞;
 3    }
 4    k[s] := 0;
 5    Q := V;
 6    while (true) do { // Endlosschleife
 7       Wähle v ∈ Q, so dass k[v] + h(v) minimal ist;
 8       if v = t then {
 9          exit; // Abbruch, sobald der Pfad zu t bekannt ist
10       }
11       Q := Q \ {v};
12       for w ∈ ℓ_v do { // ℓ_v = Menge der Nachfolgerknoten von v
13          if w ∈ Q then {
14             if k[v] + c((v, w)) < k[w] then {
15                k[w] := k[v] + c((v, w));
16                p[w] := v;
17             }
18          }
19       }
20    }
```

hier enthält $p[v]$ im Fall $v \neq s$ und $k[v] < \infty$ den entsprechenden Vorgängerverweis zur Rekonstruktion eines kürzesten Pfades.

Wenn dann die Hauptschleife im Fall $v = t$ endet (und somit t im Anschluss aus Q entfernt worden wäre), so ergibt sich die behauptete Korrektheit unmittelbar aus der zweiten Invariante.

Der Beweis der beiden Invarianten gestaltet sich recht einfach, da wir fast alle Argumente aus dem Beweis des ursprünglichen DIJKSTRA–Verfahrens übernehmen können. Der einzige Unterschied ergibt sich, wenn wir uns davon überzeugen müssen, dass ein in Zeile 7 ausgewählter Knoten endgültig optimal ist. Im DIJKSTRA–Algorithmus haben wir dazu beim Induktionsschritt von i nach $i + 1$ auf Seite 53 das folgende Argument benutzt (Zitat):

> *„Aus Q wird in Zeile 7 der Knoten v mit den geringsten Wegkosten $k[v]$ ausgewählt. Diese Kosten können nicht mehr gesenkt werden. Ist nämlich*
>
> $$s = w_0 \rightarrow w_1 \rightarrow \ldots \rightarrow w_k = v$$
>
> *ein beliebiger Pfad von s nach v, so betrachte man den ersten Knoten w_j mit*

$w_j \in Q$. Der Knoten w_j existiert wegen $w_k = v \in Q$. Ferner gilt

$$s = w_0, w_1, \ldots, w_{j-1} \in V \setminus Q \ .$$

Die erste Invariante besagt, dass die Kosten dieses Pfades von s nach w_j mindestens $k[w_j]$ betragen. Wegen der nicht–negativen Kantengewichte betragen die Gesamtkosten des Pfades

$$s = w_0 \ \rightarrow \ w_1 \ \rightarrow \ \ldots \ \rightarrow \ w_k = v$$

also ebenfalls mindestens $k[w_j]$. Nach der Wahl von v gilt aber $k[v] \leq k[w_j]$, d.h. der betrachtete Weg führt zu keiner besseren Lösung.“

Im neuen Verfahren können wir ganz ähnlich argumentieren. Zu einem Pfad

$$s = w_0 \ \rightarrow \ w_1 \ \rightarrow \ \ldots \ \rightarrow \ w_k = v$$

betrachten wir ebenfalls den ersten Knoten w_j, der in Q liegt. Die erste Invariante besagt dann wieder, dass die Kosten dieses Pfades von s nach w_j mindestens $k[w_j]$ betragen. Weiterhin betragen die Kosten auf dem Teilstück

$$w_j \ \rightarrow \ w_{j+1} \ \rightarrow \ \ldots \ \rightarrow \ w_k = v$$

mindestens $h(w_j) - h(w_k) = h(w_j) - h(v)$. Also betragen die Gesamtkosten des Pfades

$$s = w_0 \ \rightarrow \ w_1 \ \rightarrow \ \ldots \ \rightarrow \ w_k = v$$

in der Summe mindestens $k[w_j] + h(w_j) - h(v)$. Nach der Wahl von v in Zeile 7 gilt aber

$$k[v] + h(v) \leq k[w_j] + h(w_j) \ ,$$

also

$$k[v] \leq k[w_j] + h(w_j) - h(v) \ ,$$

d.h. der betrachtete Weg kann auch hier nicht billiger als $k[v]$ sein.

Da alle übrigen Argumente aus dem Korrektheitsbeweis des Dijkstra–Verfahrens nach wie vor gültig sind, ist damit alles gezeigt. □

Die entscheidende Frage ist nun, wie man eine solche Schätzfunktion vorab berechnen kann. Eine naheliegende Lösung besteht darin, einfach alle Kanten des Graphen umzuorientieren und den Dijkstra–Algorithmus von t aus zu starten. Dann können wir alle berechneten k–Werte als Funktionwerte des Schätzers übernehmen, also für alle Knoten v einfach $h(v) := k[v]$ setzen. Die Abschätzung ist dann sogar „scharf“, da alle Schätzwerte genau den kürzesten Wegekosten bis zum Zielknoten entsprechen. Allerdings kostet die so vorgenommene Berechnung der Schätzfunktion h genauso viel wie das eigentliche ursprüngliche Dijkstra–Verfahren, und nichts wäre gewonnen.

Die A^*–Strategie macht deshalb nur dann Sinn, wenn die Bereitstellung der Schätzfunktion im Verhältnis zum ursprünglichen Verfahrensaufwand nicht ins Gewicht fällt. Möglicherweise ist die Schätzfunktion schon vorab bekannt, z.B. kann sie für wichtige Verkehrsknoten im Kartenmaterial gespeichert sein.

Es gibt aber auch die Situation, dass die Berechnung der Kosten für jede Kante mit einem nicht unerheblichen Aufwand verbunden ist. In diesem Fall verursachen diese Berechnungen den eigentlichen Zeitaufwand und nicht das grundlegende Dijkstra–Verfahren selbst. Wenn sich nun die Kosten einfach abschätzen lassen, kann die Anwendung der A^*–Strategie einen erheblichen Zeitgewinn bedeuten, da dann nur für eine Auswahl von Kanten, die direkt in Richtung Ziel führen, der Berechnungsaufwand geleistet werden muss.

Ein Beispiel hierfür ist die Berechnung von Trassen für Züge auf Schienennetzen. Im Gegensatz zu Autos kann man den Zeitaufwand für eine Wegstrecke nicht einfach durch die Näherungsformel „Wegstrecke durch erlaubte Höchstgeschwindigkeit“ ermitteln (ein Güterzug braucht z.B. mehrere Minuten, um seine Höchstgeschwindigkeit von typischerweise 100 km/h zu erreichen). Für die Berechnung von Zugfahrten müssen komplizierte Differenzialgleichungen gelöst werden, die zum Teil auf numerischen Approximationsalgorithmen beruhen. Mit der obigen Näherungsformel kann man aber natürlich leicht eine untere Schranke angeben und somit die A^*–Strategie gut anwenden.

2.6. Zur Optimalität der Implementierung mit Fibonacci–Heaps

Gegen Ende des Abschnitts 2.2 auf Seite 55 wurden bereits die *Fibonacci–Heaps* als Implementierungsvariante für das Dijkstra–Verfahren erwähnt, mit deren Hilfe sich eine Laufzeit von $O(n \log n + m)$ erreichen lässt. Wir werden in diesem Abschnitt zeigen, dass eine weitere Laufzeitverbesserung nicht möglich ist.

Eine untere Schranke von $\Omega(m)$ lässt sich leicht einsehen — Verfahren, die mit weniger Zeit auskommen, können sich nicht alle Kanten des Graphen ansehen. Dann aber verpassen sie womöglich genau diejenigen Kanten, die die kürzesten Wege erzielen. Ganz offensichtlich können solchen Algorithmen daher nicht korrekt sein.

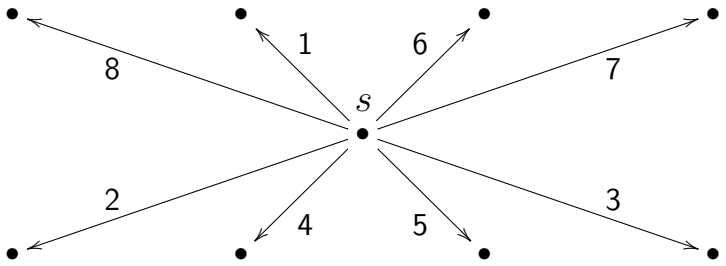
Wenn es uns nun noch gelingt, auch eine zweite untere Schranke der Größe $\Omega(n \log n)$ nachzuweisen, haben wir wegen

$$\Omega(\max\{m, n \log n\}) = \Omega(n \log n + m)$$

die Behauptung bewiesen. Wir werden dazu in zwei Schritten vorgehen:

- Wir werden zeigen, dass sich der Dijkstra–Algorithmus implizit zum Sortieren von Zahlen einsetzen lässt.
- Das Sortieren von n Zahlen benötigt (unter gewissen Voraussetzungen, die hier erfüllt sind) jedoch immer mindestens $\Omega(n \log n)$ viel Zeit.

Der erste Schritt ist einfach. Angenommen, es sind n zu sortierende Zahlen vorgegeben. Dann konstruieren wir daraus einen kantengewichteten Digraph mit $n+1$ Knoten und n Kanten, wobei ein Knoten als Startknoten ausgewählt wird (wie nennen ihn wieder s) und zentral „in der Mitte steht“. Von s aus verweist jeweils eine Kante zu allen anderen n Knoten. Jede dieser n Kanten ist mit einer der vorgegebenen Zahlen gewichtet. Sind z.B. die Zahlen 8, 1, 6, 7, 2, 4, 5 und 3 vorgegeben, so erhalten wir diesen Graph:



Nun führen wir einfach den Dijkstra–Algorithmus (siehe Seite 49) aus. Als erster Knoten wird in Zeile 7 der `while`–Hauptschleife wie immer der Startknoten s ausgewählt. In den Zeilen 9–16 verbessern sich dann die k–Werte aller übrigen Knoten auf die an den einzelnen Kanten stehenden Zahlen. Weitere Kanten gibt es nicht, somit ändern sich im weiteren Verlauf auch keine k–Werte mehr. In den folgenden Interationen der Hauptschleife werden deshalb in Zeile 7 alle übrigen n Knoten ihren k–Werten entsprechend aufsteigend sortiert ausgewählt. Wenn man diese Werte zeitgleich auf dem Bildschirm ausgibt, erhält man offenbar die ursprüngliche Zahlenfolge in sortierter Anordnung.

Die Konstruktion des Graphen mit seinen $n+1$ vielen Knoten und n Kanten ist sehr einfach und benötigt offenbar nur $O(n)$ viel Zeit. Wenn also der Dijkstra–Algorithmus in weniger als $\Omega(n \log n + m) = \Omega(n \log n)$ viel Zeit ablaufen würde, könnte man n Zahlen in insgesamt weniger als $\Omega(n \log n)$ viel Zeit sortieren. Wir werden nun zeigen, dass dies unmöglich ist.

Wenn wir einem Sortierverfahren z.B. die ersten n Zahlen in einer durcheinander gewürfelten Reihenfolge übergeben (so wie in unserem Beipiel oben), so erwarten wir, dass der Algorithmus die Zahlen neu positioniert, so dass nach allen Umpositionierungen die Zahlen in einer z.B. aufsteigend sortierten Reihenfolge vorliegen. Für $i = 1, \ldots, n$ bezeichne $\sigma(i)$ die neue Position, die von der i–ten Zahl nach der Sortierung eingenommen wird. Die „Vertauschungsvorschrift“ $\sigma : \{1, \ldots, n\} \longrightarrow \{1, \ldots, n\}$ ist also eine *Permutation*.

Die Permutation ist eindeutig bestimmt, da alle Zahlen paarweise verschieden sind. Somit muss der Algorithmus unter allen möglichen $n!$ Permutationen diese eine bestimmte Permutation finden. Grundsätzlich kann dabei jede Permutation σ die richtige sein. Beispielweise ist bei acht Zahlen die Permutation

i	1	2	3	4	5	6	7	8
$\sigma(i)$	4	6	1	7	8	5	2	3

genau dann die gesuchte Vertauschungsvorschrift, wenn die Ausgangszahlen in der Rei-

henfolge 4, 6, 1, 7, 8, 5, 2 und 3 angeordnet sind. Dadurch wird nämlich die erste Zahl (die 4) auf die vierte Position versetzt, die zweite Zahl (die 6) auf die sechste Position usw., so dass sich am Ende genau die sortierte Reihenfolge von 1 bis 8 ergibt.

Wie schnell kann der Algorithmus die gesuchte Permutation finden? Anfangs sind theoretisch alle $n!$ Kandidaten möglich. Nach einem Vergleich mittels einer `if`–Bedingung wird diese Auswahl eingeschränkt, denn manche Permutationen sind mit dem Ergebnis inkompatibel. Wenn z.B. tatsächlich acht Zahlen vorgegeben sind und der Algorithmus bei einem Vergleich feststellt, dass die zweite Zahl kleiner als die achte ist, so kann die obige Beispielpermutation sicher ausgeschlossen werden. Sie würde nämlich wegen $\sigma(2) = 6$ und $\sigma(8) = 3$ die zweite Zahl hinter die achte Zahl positionieren, d.h. die zweite Zahl müßte dann auch größer als die achte sein, was dem Vergleichsergebnis widerspricht.

Jeder Vergleich zwischen zwei Zahlen teilt deshalb die Menge der noch möglichen Permutationen in zwei Teilmengen auf. Die einen Permutationen sind mit dem positiven, die anderen mit dem negativen Vergleichsergebnis kompatibel. Da auf jede Permutation entweder das eine oder das andere zutrifft, enthält mindestens eine Teilmenge mindestens die Hälfte aller vorherigen Permutationen (ansonsten würden beide Mengen zusammen nicht die komplette vorherige Menge ergeben).

Nun kann sich die gesuchte Permutation in der größeren Teilmenge befinden. Wenn der Algorithmus mit einer für ihn möglichst ungünstigen Startreihenfolge „belastet“ wird, tritt diese Situation auch bei allen weiteren Vergleichsschritten auf. Da jedesmal die Kandidatenmenge höchstens halbiert wird, werden somit insgesamt mindestens (aufgerundet) $\log_2(n!)$ Vergleichsschritte benötigt, bis nur noch eine (die gesuchte) Permutation übrig bleibt.

Wegen

$$n! = n \cdot (n-1) \cdots 1 = \underbrace{n \cdot (n-1) \cdots}_{n\text{–mal}} \geq \underbrace{n \cdot (n-1) \cdots}_{(n/2)\text{–mal}} \geq \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdots \frac{n}{2}}_{(n/2)\text{–mal}} = \left(\frac{n}{2}\right)^{n/2}$$

folgt

$$\log_2(n!) \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2}(\log_2 n - 1) = \frac{1}{2}\, n \log_2 n - \frac{n}{2} = \Omega(n \log n)$$

und somit die behauptete untere Schranke. Also kann auch insbesondere unser „Dijkstra–Sortieralgorithmus“ nicht schneller arbeiten.

A

Das O–Kalkül

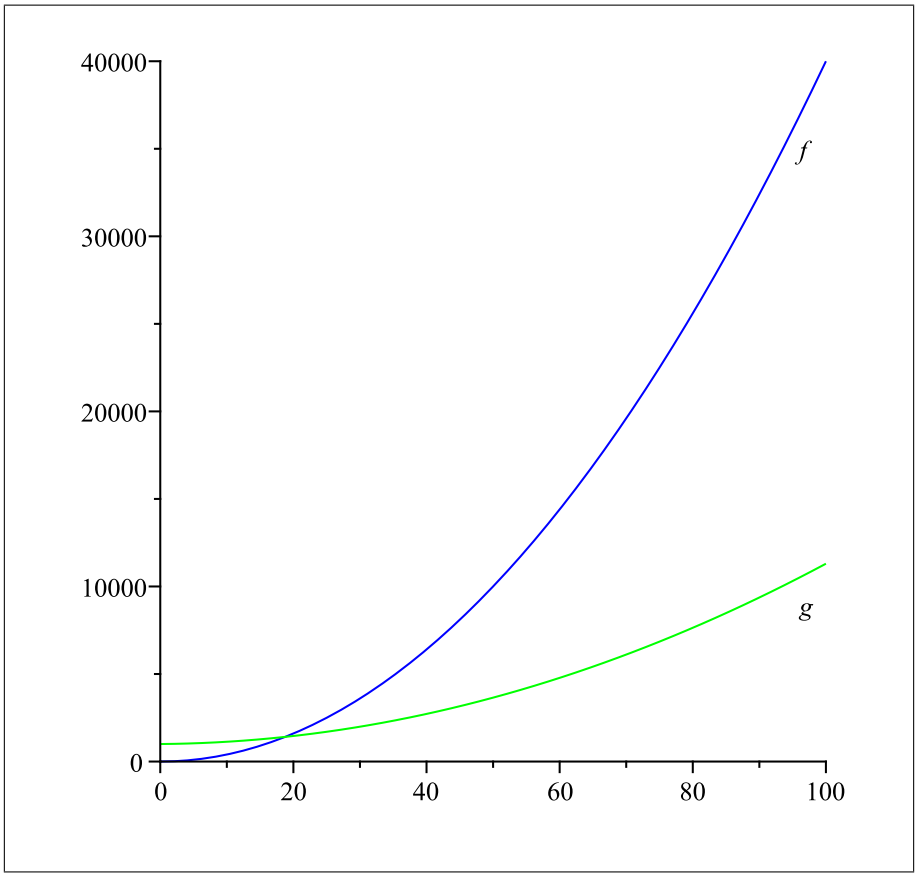
In der theoretischen Informatik wird häufig die Komplexität diverser Problemstellungen untersucht. Konkret wird dabei gefragt, wie viel Zeit ein Computer zur Ermittlung einer bestimmten Lösung benötigt. Dabei ist klar, dass eine größere Probleminstanz im Allgemeinen auch mehr Rechenzeit verlangt als eine kleinere Probleminstanz. Wenn zum Beispiel eine Menge von Zahlen sortiert werden soll, so ist dies offenbar für 20 Zahlen schneller zu erledigen als für 10000 Zahlen. Man setzt deshalb die benötigte Rechenzeit in ein Verhältnis zur Größe der konkret gestellten Aufgabe.

Angenommen, wir haben je zwei Programme F und G auf zwei (sehr langsamen) Rechnern zur Verfügung, die eine Menge von n Zahlen sortieren. Das erste Programm F benötigt dazu $f(n) := 4n^2$ Sekunden, das andere $g(n) := n^2 + 3n + 1000$ Sekunden. (Natürlich sind heutige Rechner viel schneller, aber wir bleiben der Einfachheit halber bei diesem Beispiel.) Um zu ermitteln, welches Programm das bessere (also schnellere) ist, können wir f und g für verschiedene Werte von n vergleichen:

n	$f(n)$	$g(n)$
3	36	1018
5	100	1040
10	400	1130
20	1600	1460
100	40000	11300

Alternativ kann man auch die Graphen der beiden Funktionen miteinander vergleichen, so wie Sie es aus der Mathematik sicherlich kennen. Das entsprechende Schaubild ist auf der nächsten Seite zu finden.

Anfangs ist also F vermutlich das bessere Programm, weil es weniger Zeit als G benötigt. Für größere Werte von n (genauere Untersuchungen zeigen: ab $n = 19$) ist aber G schneller, und wie man an den Tabellenwerten bzw. den Funktionsgraphen erkennt, wird der Abstand zwischen $f(n)$ und $g(n)$ mit größer werdenden Argumenten für n immer



deutlicher. Ist also G das bessere Programm?

Zunächst einmal ist es so, dass man sich für kleinere Probleminstanzen in der Informatik nicht wirklich interessiert (3, 5 oder 10 Zahlen kann man ja auch noch von Hand sortieren, eine Million Zahlen aber nicht). Deshalb ist immer entscheidend, wie sich die Rechenzeiten für große Instanzen entwickeln. Der Begriff „groß“ ist dabei dehnbar: das Sortieren von 1000 Zahlen ist für Menschen bereits eine sehr langwierige Aufgabe, aber für moderne Computer ist diese Größenordnung selbst bei der Wahl von schlechten Algorithmen eine Angelegenheit von Millisekunden. Man hat sich deshalb darauf geeinigt, dass man einfach eine beliebig große (aber endliche) Anzahl von Fällen am Anfang ausschließt und erst danach mit dem Vergleich beginnt. Also ist so gesehen G tatsächlich das bessere Programm.

Trotzdem muss aber paradoxerweise G nicht unbedingt besser als F sein. Wegen

$$f(n) = 4n^2 \leq 4n^2 + 12n + 4000 \leq 4(n^2 + 3n + 1000) = 4 \cdot g(n)$$

gilt nämlich $f(n) \leq 4 \cdot g(n)$, d.h. F ist höchstens viermal langsamer als G, und diese Tatsache muss nicht unbedingt etwas mit der Güte des Verfahrens von F zu tun haben.

Vielleicht läuft F ja auf einem älteren Computer, oder ist in einer „langsameren“ Programmiersprache geschrieben worden. Konstante Faktoren sind daher keine verlässliche Aussage für den Vergleich von Algorithmen und werden deshalb vernachlässigt.

Unter diesen Gesichtpunkten kann man die Angabe der „genauen“ Zeitangaben nun vereinfachen. Wir führen dies gleich am Beispiel unserer zweiten *Komplexitätsfunktion* $g(n) = n^2 + 3n + 1000$ vor. Ziel ist es, für $g : \mathbb{N}_0 \longrightarrow \mathbb{N}_0$ eine vereinfachende Abschätzung $h : \mathbb{N}_0 \longrightarrow \mathbb{R}$ zu finden, die eine *asymptotische obere Schranke* von g darstellt ($\mathbb{R}$ ist dabei die Menge der *reellen Zahlen*). Die Funktion h soll also der Zeitverbrauch g nach oben hin begrenzen, d.h. das Programm soll für alle Eingabegrößen n nicht mehr als $h(n)$ viel Zeit verbrauchen. Es soll also

$$\forall n \in \mathbb{N}_0 \colon\ g(n) \leq h(n)$$

gelten. Allerdings besagt unsere erste Vereinbarung, dass wir endlich viele Werte am Anfang ignorieren können. Man darf also einen Startwert $n_0 \in \mathbb{N}$ wählen und betrachtet die Vergleichsbedingung nur für alle Argumente n, die größer oder gleich n_0 sind:

$$\exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq h(n)\ .$$

Weiterhin haben wir gesehen, dass konstante Faktoren $c \in \mathbb{N}$ im Geschwindigkeitsvergleich keine Rolle spielen. Es ist beispielsweise egal, ob wir Sekunden oder Minuten messen, da sich beide Einheiten nur durch einen konstanten (also unwichtigen) Faktor von 60 unterscheiden. Wir sind also schon zufrieden, wenn

$$\exists c \in \mathbb{N} \colon\ \exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)$$

gilt und schreiben dann $g = O(h)$ gemäß der folgenden Definition.

Definition Sei $h : \mathbb{N}_0 \longrightarrow \mathbb{R}$ eine Funktion. Mit $O(h)$ (sprich: „groß Oh von h“ oder einfach „Oh von h“) bezeichnen wir die Menge aller Funktionen, die — abgesehen von einem konstanten Faktor c und endlich vielen Ausnahmen $n_0 \in \mathbb{N}$ am Anfang — *höchstens* so schnell wachsen wie h:

$$O(h) := \{g : \mathbb{N}_0 \longrightarrow \mathbb{R} \mid \exists c \in \mathbb{N} \colon\ \exists n_0 \in \mathbb{N} \colon\ \forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)\}\ .$$

Für $g \in O(f)$ sagt man auch: „g ist von der Ordnung f“.

Statt $g \in O(f)$ schreibt man üblicherweise $g = O(f)$, aber dies ist nur eine Notation und keine wirkliche Gleichung (denn auf der linken Seite steht eine einzelne Funktion, und auf der rechten Seite eine Menge von Funktionen). Man darf beide Seiten auch nicht tauschen, d.h. die Notation $O(f) = g$ ist unzulässig.

Beispiel Als erstes betrachten wir die Funktion $g(n) = n^2 + 3n + 1000$ vom Anfang dieses Abschnitts. Dann können wir Folgendes beobachten:

- Für alle $n \geq 3$ ist $3n \leq n^2$.

- Für alle $n \geq 32$ ist $1000 \leq n^2$.
- Wir nehmen nun das Maximum von 3 und 32 und wissen dann, dass für alle $n \geq 32$ beide obigen Abschätzungen richtig sind. Für alle $n \geq 32$ ist also der Funktionswert $g(n) = n^2 + 3n + 1000$ kleiner oder gleich $n^2 + n^2 + n^2 = 3n^2$.

Mit der Wahl $c := 3$, $n_0 := 32$ und $h(n) := n^2$ ist demnach die Abschätzung

$$\forall n \geq n_0 \colon\ g(n) \leq c \cdot h(n)$$

richtig. Also gilt $g = O(n^2)$.

An dem vorstehenden Beispiel erkennt man gut, wie die O–Notation in der Praxis funktioniert. Man hat es bei einer exakten Laufzeitangabe üblicherweise mit einer Summe von verschiedenen Termen zu tun (im Beispiel waren dies n^2, $3n$, und 1000). Die O–Notation sucht sich nun im Endeffekt unter diesen Termen den am schnellsten wachsenden heraus (dies war hier n^2). Denn durch eine genügend große Wahl von n_0 (hier 32) dominiert dieser Term für $n \geq n_0$ alle anderen, so dass ein Vielfaches davon (im Beispiel war es das Dreifache) eine obere Schranke des ursprünglichen Funktionswertes darstellt. Der zugehörige konstante Faktor kann dann abschließend durch eine entsprechende Wahl der Konstaten c ausgeglichen werden.

Auch bzgl. der anderen Funktion $f(n) = 4n^2$ erhalten wir $f = O(n^2)$. Zum Beweis könnte man wie im letzten Beispiel formal $c := 4$, $n_0 := 1$ und $h(n) := n^2$ wählen und dann zeigen, dass immer

$$\forall n \geq n_0 \colon\ f(n) \leq c \cdot h(n)$$

gilt. Einfacher ist jedoch die Argumentation mit der unter dem Beispiel angedeuteten Methode: es gibt hier sowieso nur einen Term, konstante Faktoren können fallen gelassen werden — fertig.

Sowohl f als auch g sind also von sog. *quadratischer Komplexität*, und damit sind die beiden zugrundeliegenden Programme F und G im Endeffekt gleich gut.

Wir können das für $g(n) = n^2 + 3n + 1000$ vorgebrachte Argument verallgemeinern:

Satz Sei $f : \mathbb{N}_0 \to \mathbb{R}$ ein *Polynom* vom *Grad* k, d.h. für passende (evtl. negative) *Koeffizienten* $a_0, a_1, \ldots, a_k \in \mathbb{R}$ ist f von der Form

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

bzw. (in Summenschreibweise)

$$f(n) = \sum_{i=0}^{k} a_i n^i \ .$$

(Für das Polynom $n^2 + 3n + 1000$ wäre z.B. $a_0 = 1000$, $a_1 = 3$ und $a_2 = 1$.)

Dann gilt $f = O(n^k)$.

Beweis: Zunächst verändern wir das Polynom und wandeln alle negativen Koeffizienten in positive um (aus z.B. $7n^3+17n^2-23n+2$ wird so $7n^3+17n^2+23n+2$.) Das modifizierte

Polynom ist dann offenbar eine obere Schranke der ursprünglichen Funktion, da alle Terme, die vorher abgezogen wurden, jetzt addiert werden. Ein verbleibender Term der Form $a_i n^i$ wächst nun umso schneller, je größer der Exponent i ist. Folglich ist n^k der dominierende Term. □

Es gilt also z.B.

$$7n^3 + 17n^2 - 23n + 2 = O(n^3) \ .$$

Natürlich gilt theoretisch auch $7n^3 + 17n^2 - 23n + 2 = O(n^4)$, denn n^4 wächst stärker als n^3, d.h. n^4 ist erst recht eine asymptotische obere Schranke. (Beachten Sie, dass man das Polynom auch als $0n^4 + 7n^3 + 17n^2 - 23n + 2$ schreiben könnte, womit die eben bewiesene Regel wieder direkt anwendbar wäre.) Wir werden jedoch später sehen, warum man trotzdem normalerweise immer $7n^3 + 17n^2 - 23n + 2 = O(n^3)$ schreibt.

Die O–Notation kommt nicht nur bei Polynomen zum Einsatz, sondern auch z.B. bei der Vereinfachung logarithmischer Terme. (Zur Erinnerung: $\log_b y$ bezeichnet den *Logarithmus* von y zur *Basis* b, d.h. $\log_b y$ ist diejenige Zahl x, für die $b^x = y$ gilt. Beispielsweise ist $\log_2 8 = 3$ und $\log_{\sqrt{2}} 4 = 4$.) Im Zusammenhang mit der O–Notation können wir nun bei der Verwendung von Logarithmen die Basisangaben stets weglassen. Denn für zwei Logarithmen $\log_a n$ und $\log_b n$ gilt immer $\log_a n = \log_a b \cdot \log_b n$, und die Größe $\log_a b$ ist eine Konstante. Diese wiederum ist in der O–Notation irrelevant, d.h. es gilt $log_a n = O(log_b n)$ und $\log_b n = O(\log_a n)$. Wir schreiben zukünftig einfach nur noch $O(\log n)$.

Bei der Vereinfachung von Laufzeiten durch die O–Notation treten bestimmte *Komplexitätsklassen* wie z.B. das obige $O(n^2)$ öfter auf. Folgende Klassen werden häufig benutzt:

Ordnung	Wachstum
$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n \log n)$	„*n–log–n*“
$O(n^2)$	quadratisch
$O(n^3)$	kubisch
$O(n^k)$ für ein $k > 1$	polynomiell
$O(d^n)$ für ein $d > 1$	exponentiell

Die Tabelle ist dabei nach steigendem Zeitaufwand sortiert. (Ausnahme: für $1 < k < 2$ liegt die Komplexität von $O(n^k)$ zwischen $O(n \log n)$ und $O(n^2)$, für $2 < k < 3$ liegt sie zwischen $O(n^2)$ und $O(n^3)$.) Da jeder O–Ausdruck eine Menge von Funktionen darstellt, kann man die wachsende Komplexität auch durch eine Kette von echten Teilmengenbeziehungen ausdrücken (dabei sind wie oben d und k Zahlen, die größer als Eins sind, wohingegen ε eine „kleine“ Zahl bezeichnet, die echt zwischen 0 und 1 liegt):

$$O(1) \subset O(\log n) \subset O(n^\varepsilon) \subset O(n) \subset O(n \log n) \subset O(n^k) \subset O(d^n) \ .$$

Für $\ell > 0$ gilt zudem

$$O(n^\ell) \subset O(n^{\ell+\varepsilon})$$

sowie für $d > 1$

$$O(d^n) \subset O((d+\varepsilon)^n) \ .$$

Besonders die zweite Regel führt zuweilen zur Verwirrung. Es gilt z.B. *nicht* $3^n = O(2^n)$, obwohl sich die beiden Funktionen scheinbar nur durch verschiedene führende Zahlen unterscheiden. Die Zahlen 2 und 3 sind hier jedoch keine multiplikativen Konstanten, sondern gehen als unterschiedliche Basen in die zugehörigen Exponentialfunktionen ein. Man kann leicht nachprüfen, dass 3^n viel schneller als 2^n wächst. Dies kann auch nicht durch eine noch so große multiplikative Konstante c ausgeglichen werden. Egal, wie man c auch wählt: für genügend große n gilt stets $3^n > c \cdot 2^n$.

Wie „gut“ ist nun ein Programm, wenn es eine bestimmte (z.B. quadratische) Komplexität hat? Dies kann man sich anhand der Auswirkungen auf die Laufzeit klarmachen, wenn man die Eingabegröße verändert. Hierzu einige Beispiele:

- Bei quadratischer Komplexität führt die *Verdopplung* der Eingabegröße n (wenn man also z.B. doppelt so viele Zahlen sortiert) zu einer *Vervierfachung* der Laufzeit, nämlich von n^2 auf $(2n)^2 = 4n^2$.
- Bei logarithmischer Komplexität sieht dies viel besser aus: hier führt die Verdopplung der Eingabegröße n nur zu konstant vielen Zusatzschritten, da z.B. für den Zweierlogarithmus $\log_2 2n = \log_2 n + 1$ gilt.
- Umgekehrt sind bei exponentieller Komplexität selbst die schnellsten Rechner der Welt machtlos, denn schon eine Vergrößerung der Eingabe um nur wenige neue Elemente führt zu einer Verdopplung der Laufzeit. Kann z.B. ein Computer ein bestimmtes Problem für 1000 Eingabeelemente in nur einer Mikrosekunde lösen, so dauert bei einer Zeitkomplexität von $O(2^n)$ die Lösung des gleichen Problems für 1050 Elemente schon knapp 36 Jahre (!), obwohl sich die Eingabe nur um 5% vergrößert hat. Solche Probleme werden daher häufig als „hartnäckig“ bezeichnet.

Jetzt wird auch klar, warum man $7n^3 + 17n^2 - 23n + 2$ zu $O(n^3)$ und nicht zu $O(n^4)$ vereinfacht. Denn die Laufzeit eines Verfahrens ist ein zentrales „Gütesiegel“ — je größer die Laufzeit, desto schlechter das Verfahren. Also versucht man die durch die O–Notation vorgenommene obere Abschätzung möglichst weit nach unten zu drücken, und durch die Angabe $7n^3 + 17n^2 - 23n + 2 = O(n^4)$ würde man freiwillig das zentrale Bewertungskriterium verschlechtern.

Neben der O–Notation gibt es noch weitere sog. *Landausche*[1] *Symbole*. Wir werden vor allem noch das Ω–Symbol verwenden. Die Idee und Verwendung der Ω–Notation ist dabei im Prinzip genau dieselbe wie bei der O–Notation, jedoch verwendet man das Ω–Symbol bei asymptotischen *unteren* Schranken. Wenn also ein Programm in $\Omega(n^2)$

[1] Edmund Landau, *1877 Berlin, †1938 Berlin, deutscher Mathematiker, ab 1909 Professor für reine Mathematik in Göttingen.

viel Zeit abläuft, so drückt man damit aus, dass es *mindestens* (und nicht wie bei der O–Notation *höchstens*) quadratisch viele Rechenschritte benötigt.

Zum Abschluss sei noch bemerkt, dass man mit der O–und Ω–Notation nicht nur Laufzeiten beschreiben kann, sondern auch z.B. den Verbrauch von Speicherplatz. Man spricht dann auch von der *Platzkomplexität* eines Programms.

B

Beweisführungen durch vollständige Induktion

Im Verlauf der Vorlesung werden mehrmals Beweise mit Hilfe der sogenannten *vollständigen Induktion* geführt. Diese Beweistechnik eignet sich für vielen Behauptungen über natürliche Zahlen oder einen (unendlichen) Teilbereich davon. Betrachten Sie z.B. folgende Aussage. Sie besagt, dass die Summe der ersten n ungeraden Zahlen gleich der n–ten Quadratzahl ist:

$$1 + 3 + 5 + \cdots + (2 \cdot n - 1) = n^2 \ .$$

Abkürzend kann man diese Gleichung auch mit Hilfe der Summendarstellung

$$\sum_{i=1}^{n} (2i - 1) = n^2$$

notieren. Hierbei ist i wieder ein Index, der alle ganzen Zahlen zwischen den beiden Grenzen durchläuft, die unter und über dem großen Sigma–Symbol angegeben sind (hier also 1 und n). Der jeweilige Wert wird für alle Vorkommen von i in dem Term hinter dem Sigma–Symbol eingesetzt, und die resultierenden Ergebnisse werden aufaddiert. Hier ergibt sich also wie gewünscht die Summe aus den Werten $2 \cdot 1 - 1 = 1, 2 \cdot 2 - 1 = 3$, usw.

Wir können die Behauptung durch direktes Nachrechnen probeweise überprüfen:

$$n = 1: \quad \sum_{i=1}^{1} (2i - 1) = 1 = 1^2$$

$$n = 2: \quad \sum_{i=1}^{2} (2i - 1) = 1 + 3 = 4 = 2^2$$

$$n = 3: \quad \sum_{i=1}^{3} (2i - 1) = 1 + 3 + 5 = 9 = 3^2$$

$$
\begin{aligned}
n = 4: &\quad \sum_{i=1}^{4}(2i-1) = 1+3+5+7 = 16 = 4^2 \\
n = 5: &\quad \sum_{i=1}^{5}(2i-1) = 1+3+5+7+9 = 25 = 5^2 \\
n = 6: &\quad \sum_{i=1}^{6}(2i-1) = 1+3+5+7+9+11 = 36 = 6^2 \\
&\quad \dots
\end{aligned}
$$

Die Formel scheint also richtig zu sein. Die obigen „Tests“ sind aber kein Beweis, denn vielleicht stimmt ja die Aussage für $n = 7$ schon nicht mehr. Andererseits können wir natürlich auch nur endlich viele Tests vornehmen. Die obige Behauptung besteht aber in Wirklichkeit aus unendlich vielen Behauptungen, nämlich je einer Aussage für jede konkrete natürliche Zahl n. Durch direktes Nachrechnen kann der obige Satz also nicht gezeigt werden.

Die Beweistechnik der vollständigen Induktion besteht aus zwei separat zu beweisenden Teilen:

- Als erstes ist die Behauptung für $n = 1$ zu beweisen. Diesen Schritt nennt man die sogenannte *Induktionsbasis* oder *Induktionsverankerung*, manchmal auch den *Induktionsanfang*. Meistens ist dieser Schritt sehr einfach.
- Anschließend ist der *Induktionsschritt* durchzuführen. Für alle $n \in \mathbb{N}$ ist hierfür zu zeigen: falls die Behauptung bereits für n wahr ist (die sogenannte *Induktionsannahme* bzw. *Induktionshypothese*), so ist sie auch für den Fall $n + 1$ richtig.

Hat man beide Teile gezeigt, so ist die Aussage für alle $n \in \mathbb{N}$ gültig. Denn zunächst einmal gilt sie für $n = 1$, was man in der Induktionsverankerung nachgewiesen hat. Da die Aussage für $n = 1$ gilt, gilt sie durch Ausnutzung des Induktionsschritts dann auch für den Fall $n = 1 + 1 = 2$. Dann aber gilt sie auch für den Fall $n = 3$ durch erneute Ausnutzung des Induktionsschritts. Genauso kann man nun weiter für $n = 4, 5, 6, \dots$ argumentieren, d.h. die Aussage gilt für alle $n \in \mathbb{N}$.

Wir wenden diese Technik jetzt für das obige Beispiel an.

Induktionsbasis: Für $n = 1$ war, wie bereits gesehen, die Aussage richtig ($1 = 1^2$).

Induktionsschritt: n sei nun eine beliebige natürliche Zahl. Wir können aufgrund der Induktionshypothese annehmen, dass die Aussage für diese konkrete Zahl n bereits bewiesen ist. Es gilt also:

$$\sum_{i=1}^{n}(2i-1) = n^2 \ .$$

Wir müssen zeigen, dass sie dann auch für den Fall $n + 1$ stimmt. Hierfür ersetzen wir überall das Vorkommen von „n“ durch „$n + 1$“, also

$$\sum_{i=1}^{n+1}(2i-1) = (n+1)^2 \ .$$

Zum Beweis dieser Aussage kann man nun die Induktionshypothese geschickt ausnutzen. Es gilt nämlich

$$\begin{aligned}
&\sum_{i=1}^{n+1}(2i-1)\\
=\ &\Big(\sum_{i=1}^{n}(2i-1)\Big)+(2\cdot(n+1)-1)\\
=\ &n^2+(2\cdot(n+1)-1) \qquad \text{(Verwendung der Induktionshypothese)}\\
=\ &n^2+2n+2-1\\
=\ &n^2+2n+1\\
=\ &(n+1)^2
\end{aligned}$$

Der induktive Beweis ist damit vollständig geführt. □

Man kann beim Induktionsstart auch eine andere Basis als $n = 1$ wählen, z.B. $n = 0$. Dann gilt die Aussage entsprechend für alle $n \in \mathbb{N}_0$ statt nur für $n \in \mathbb{N}$. Genauso darf man beim Induktionschritt für den Fall $n+1$ nicht nur den Fall n als Induktionshypothese zurückgreifen, sondern auch alle vorherigen Fälle $n-1, n-2, \ldots, 1$, die ja ebenfalls nach dem obigen Prinzip bereits wahr sein müssen.

Beispiel Wir zeigen durch vollständige Induktion über $n \in \mathbb{N}_0$:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \ .$$

Diese Formel wird z.B. auf Seite 54 bei der Abschätzung der Laufzeit von dem DIJKSTRA–Algorithmus benutzt.

Induktionsbasis: Im Fall $n = 0$ ist die Aussage wegen

$$\sum_{i=1}^{0} i = 0 = \frac{0(0+1)}{2}$$

richtig.

Induktionsschritt: Für den Induktionsschritt $n \to n+1$ folgern wir weiter

$$\begin{aligned}
\sum_{i=1}^{n+1} i &= \Big(\sum_{i=1}^{n} i\Big) + (n+1) = \frac{n(n+1)}{2} + n + 1\\
&= \frac{n(n+1)+2n+2}{2} = \frac{n^2+3n+2}{2} = \frac{(n+1)(n+2)}{2} \ .
\end{aligned}$$

Also ist die Aussage auch für den Fall $n+1$ richtig und damit vollständig bewiesen. Beachten Sie, dass wir am Ende der ersten Zeile wieder die Induktionshypothese verwendet haben.

Beispiel Die *Fibonacci–Zahlen* (sprich: „Fiehbohnatschi“) sind eine unendliche Folge von Zahlen $F_0, F_1, F_2, \ldots$ und wie folgt definiert:

$$F_n := \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ F_{n-1} + F_{n-2} & \text{falls } n \geq 2 \end{cases}$$

Es gilt also $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, usw. Wir beweisen durch vollständige Induktion die folgende Formel, mit der man direkt die n–te Fibonacci–Zahl ausrechen kann:

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] .$$

Hierzu benötigen wir zunächst die beiden Identitäten:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = 1 + \left(\frac{1+\sqrt{5}}{2}\right) \qquad \text{und} \qquad \left(\frac{1-\sqrt{5}}{2}\right)^2 = 1 + \left(\frac{1-\sqrt{5}}{2}\right) .$$

Diese sind wegen

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = \left(\frac{1}{2}+\frac{\sqrt{5}}{2}\right)^2 = \frac{1}{4}+2\cdot\frac{1}{2}\cdot\frac{\sqrt{5}}{2}+\frac{5}{4} = \frac{6}{4}+\frac{\sqrt{5}}{2} = 1+\frac{1}{2}+\frac{\sqrt{5}}{2} = 1+\left(\frac{1+\sqrt{5}}{2}\right)$$

und

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = \left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^2 = \frac{1}{4}-2\cdot\frac{1}{2}\cdot\frac{\sqrt{5}}{2}+\frac{5}{4} = \frac{6}{4}-\frac{\sqrt{5}}{2} = 1+\frac{1}{2}-\frac{\sqrt{5}}{2} = 1+\left(\frac{1-\sqrt{5}}{2}\right)$$

korrekt.

Nun prüft man für den Induktionsstart die beiden Fälle $n = 0$ und $n = 1$ nach. Im Gegenzug werden wir dann beim Induktionsschritt für $n + 1$ auf die beiden vorherigen Fälle $n - 1$ und n als Induktionshypothesen zurückgreifen können. Tatsächlich gilt

$$F_0 = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0\right] = \frac{1}{\sqrt{5}}(1-1) = 0$$

sowie

$$F_1 = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1\right]$$

$$= \frac{1}{\sqrt{5}}\left[\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right] = \frac{1}{\sqrt{5}}\left[\frac{1}{2}+\frac{\sqrt{5}}{2}-\frac{1}{2}+\frac{\sqrt{5}}{2}\right] = \frac{1}{\sqrt{5}}\left[2\cdot\frac{\sqrt{5}}{2}\right] = 1 .$$

Für $n = 0$ und $n = 1$ ist die Formel also richtig. Für den Induktionsschritt nach $n + 1$ können wir dann wie bereits erwähnt annehmen, dass die Formel für den Fall $n - 1$ und n bereits gültig ist, d.h. wie dürfen die Gleichungen

$$F_{n-1} = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]$$

und

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] .$$

als gültig voraussetzen. Wir addieren nun diese beiden Gleichungen zusammen. Auf der linken Seite ergibt sich wegen $F_{n-1} + F_n = F_{n+1}$ gerade der Wert F_{n+1}, und für die rechten Seite erhalten wir

$$\begin{aligned}
& \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right] + \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} + \left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\left(1+\left(\frac{1+\sqrt{5}}{2}\right)\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\left(1+\left(\frac{1-\sqrt{5}}{2}\right)\right)\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\left(\frac{1+\sqrt{5}}{2}\right)^{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\left(\frac{1-\sqrt{5}}{2}\right)^{2}\right] \\
= & \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] .
\end{aligned}$$

(Zwischen der vierten und fünften Zeile haben wir dabei die vorab gezeigten Identitäten ausgenutzt.) Insgesamt gilt also

$$F_{n+1} = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] ,$$

und dies ist gerade die zu beweisende Formel für den Fall $n + 1$.

In den meisten Beweisen werden der Induktionsstart und der Induktionsschritt nicht wie oben explizit herausgestellt, sondern die benötigten Argumente für die beiden Beweisteile werden in einem fließenden Text einfach hintereinander geschrieben. Auch die Verwendung der Induktionsannahme wird nicht besonders erwähnt. Betrachten Sie dazu die folgenden beiden abschließenden Beispiele.

Beispiel Wir zeigen, dass jede dritte Fibonacci–Zahl $F_0, F_3, F_6, \ldots$ (also die Folge F_{3n} für $n \in \mathbb{N}_0$) immer eine gerade Zahl ist. Für $n = 0$ ist nämlich $F_0 = 0$ und damit eine gerade Zahl. Weiterhin gilt

$$F_{3(n+1)} = F_{3n+3} = F_{3n+2} + F_{3n+1} = F_{3n+1} + F_{3n} + F_{3n+1} = 2 \cdot F_{3n+1} + F_{3n} .$$

Nun ist $2 \cdot F_{3n+1}$ sicherlich ebenso eine gerade Zahl wie F_{3n}(Induktionshypothese!), so dass sich $F_{3(n+1)}$ also aus einer Summe von zwei geraden Zahlen zusammensetzt. Demnach muss $F_{3(n+1)}$ ebenfalls gerade sein.

Beispiel Für alle $n \in \mathbb{N}_0$ ist $5^n + 3$ durch 4 teilbar.

Beweis: Im Fall $n = 0$ gilt $5^n + 3 = 1 + 3 = 4$, und 4 ist durch 4 teilbar. Ferner gilt

$$5^{n+1} + 3 = 5 \cdot 5^n + 3 = 5 \cdot 5^n + 15 - 12 = 5 \cdot (5^n + 3) - 12 \ ,$$

und da sowohl 12 als auch $5^n + 3$ durch 4 teilbar sind, ist auch $5 \cdot (5^n + 3) - 12$ durch 4 teilbar.

Nicht immer ist die Variable n, über die man die Induktion führen möchte, direkt in der zu beweisenden Aussage enthalten. Man muss sie sich dann erst „geschickt“ wählen. Dies gilt vor allem für sog. *strukturelle Induktionen*, die sich auf den Aufbau von bestimmten Objekten beziehen. So formuliert z.B. Satz 1.2.6 auf Seite 21 eine Aussage, bei der ein Weg zwischen zwei Knoten eine Rolle spielt. Bewiesen wird die Aussage dann durch Induktion über die Länge ℓ dieses Weges.

Literaturverzeichnis

[1] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction To Algorithms*, 2nd Edition, MIT Press, 2001.

[2] R. H. Güting, St. Dieker, *Datenstrukturen und Algorithmen*, 3. Auflage, Vieweg+Teubner, 2004.

[3] Th. Ottmann, P. Widmayer, *Algorithmen und Datenstrukturen*, 4. Auflage, Spektrum Akademischer Verlag, 2002.

Index