Skript zur Vorlesung

Algorithmen und Berechnungskomplexität I

Prof. Dr. Heiko Röglin
Institut für Informatik I

Wintersemester 2010/11

7. Januar 2011

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `roeglin@cs.uni-bonn.de`.

Kapitel 4

Elementare Graphalgorithmen

4.3 Minimale Spannbäume

Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph. Sei $w : E \to \mathbb{R}_{>0}$ eine *Gewichtung der Kanten*, d.h. eine Funktion, die jeder Kante $e \in E$ ein Gewicht $w(e)$ zuordnet.

Definition 4.3.1. *Eine Kantenmenge $T \subseteq E$ heißt* Spannbaum *von G, wenn der Graph $G = (V, T)$ zusammenhängend und kreisfrei ist. Wir erweitern die Funktion w auf Teilmengen von E, indem wir definieren*

$$w(T) = \sum_{e \in T} w(e),$$

und wir nennen $w(T)$ das Gewicht von T. *Ein Spannbaum $T \subseteq E$ heißt* minimaler Spannbaum *von G, wenn es keinen Spannbaum von G gibt, der ein kleineres Gewicht als T hat.*

Die Berechnung eines minimalen Spannbaumes ist ein Problem, das in vielen Anwendungskontexten auftritt. Wir entwerfen in diesem Abschnitt einen Algorithmus, der effizient für einen gegebenen Graphen, einen minimalen Spannbaum berechnet.

4.3.1 Union-Find-Datenstrukturen

Um diesen Algorithmus beschreiben zu können, benötigen wir zunächst eine sogenannte *Union-Find-* oder *Disjoint-Set-Datenstruktur*. Diese Datenstruktur dient zur Verwaltung einer Menge von disjunkten Mengen $S_1, \ldots, S_k$. Für jede von diesen disjunkten Mengen S_i gibt es einen beliebigen Repräsentanten $s_i \in S_i$. Die Datenstruktur muss die folgenden Operationen unterstützen:

- MAKE-SET(x): Erzeugt eine neue Menge $\{x\}$ mit Repräsentant x. Dabei darf x nicht bereits in einer anderen Menge vorkommen.
- UNION(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.
- FIND(x): Liefert den Repräsentanten der Menge S mit $x \in S$ zurück.

Im folgenden Beispiel soll $x : A$ bedeuten, dass wir eine Menge A mit Repräsentant x gespeichert haben. Welches Element nach einer UNION-Operation Repräsentant der neuen Menge wird, ist nicht eindeutig bestimmt. Es wurde jeweils ein beliebiges Element dafür ausgewählt.

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1 : \{1\}$
MAKE-SET(2), MAKE-SET(3)	$1 : \{1\}$, $2 : \{2\}$, $3 : \{3\}$
MAKE-SET(4), MAKE-SET(5)	$1 : \{1\}$, $2 : \{2\}$, $3 : \{3\}$, $4 : \{4\}$, $5 : \{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2 : \{1, 2\}$, $3 : \{3\}$, $4 : \{4\}$, $5 : \{5\}$
UNION(3,4)	$2 : \{1, 2\}$, $3 : \{3, 4\}$, $5 : \{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2 : \{1, 2, 5\}$, $3 : \{3, 4\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(5,4)	$3 : \{1, 2, 3, 4, 5\}$

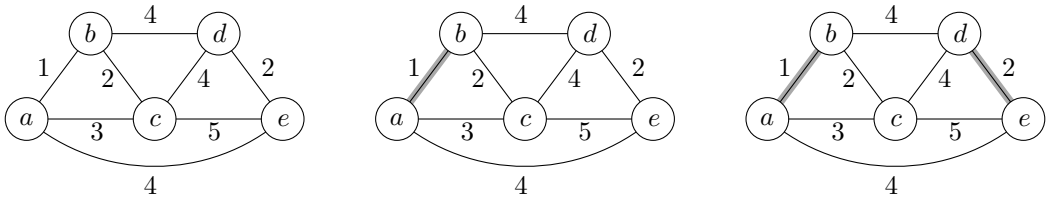
4.3.2 Algorithmus von Kruskal

Mit Hilfe einer solchen Union-Find-Datenstruktur können wir nun den *Algorithmus von Kruskal* zur Berechnung eines minimalen Spannbaumes angeben. Er ist nach Joseph Kruskal benannt, der diesen Algorithmus 1956 veröffentlichte.

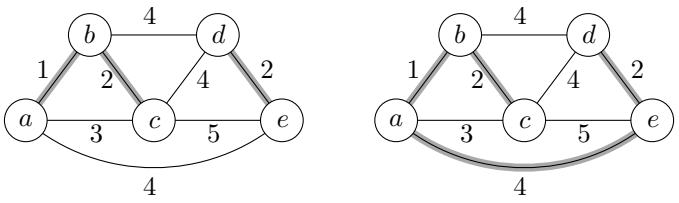
```
    KRUSKAL(G, w)
1   Teste mittels DFS, ob G zusammenhängend ist. Falls nicht Abbruch.
2   for all v ∈ V { MAKE-SET(v) }
3   T = ∅
4   Sortiere die Kanten in E gemäß ihrem Gewicht. Danach gelte E = {e_1, ..., e_m} mit
    w(e_1) ≤ w(e_2) ≤ ... ≤ w(e_m). Außerdem sei e_i = (u_i, v_i).
5   for i = 1 to m {
6       if FIND(u_i) ≠ FIND(v_i) {
7           T = T ∪ {e_i}
8           UNION(u_i, v_i)
9       }
10  }
11  return T
```

Die Mengen, die in der Union-Find-Datenstruktur im Algorithmus verwendet werden, entsprechen den Zusammenhangskomponenten des Graphen $G = (V, T)$. Wir gehen die Kanten in der Reihenfolge ihres Gewichtes durch und fügen eine Kante zur Menge T hinzu, wenn sie nicht innerhalb einer bereits zusammenhängenden Menge verläuft.

Bevor wir die Korrektheit beweisen und die Laufzeit analysieren, demonstrieren wir das Verhalten an einem kleinen Beispiel.



$\{a\},\{b\},\{c\},\{d\},\{e\}$ $\{a,b\},\{c\},\{d\},\{e\}$ $\{a,b\},\{c\},\{d,e\}$



$\{a,b,c\},\{d,e\}$ $\{a,b,c,d,e\}$

Korrektheit

Die Korrektheit des Algorithmus folgt aus dem folgenden Lemma.

Lemma 4.3.2. *Sei $V_1,\ldots,V_k$ eine disjunkte Zerlegung von V. Sei $E_i \subseteq E$ die Menge der Kanten $(u,v) \in E$ mit $u,v \in V_i$ und sei $T_i \subseteq E_i$ ein Spannbaum des Graphen (V_i,E_i).*

Ferner sei $E' = E \setminus (E_1 \cup \ldots \cup E_k)$, also

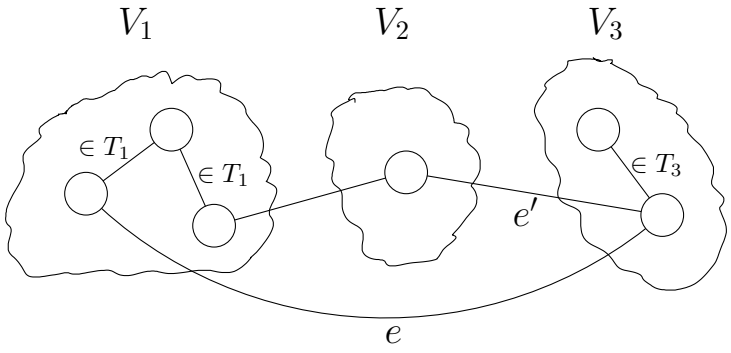
$$E' = \{(u,v) \in E \mid u \in V_i, v \in V_j, i \neq j\},$$

und sei e eine Kante mit dem kleinsten Gewicht aus E'.

Unter allen Spannbäumen von $G = (V,E)$, die die Kanten aus $T_1 \cup \ldots \cup T_k$ enthalten, gibt es einen mit kleinstem Gewicht, der zusätzlich auch die Kante e enthält.

Beweis. Sei T ein Spannbaum von $G = (V,E)$, der $T_1 \cup \ldots \cup T_k$ aber nicht e enthält. Wir konstruieren einen Spannbaum T', der $T_1 \cup \ldots \cup T_k$ und e enthält und für den gilt $w(T') \leq w(T)$. Daraus folgt direkt das Lemma, denn angenommen wir haben einen Spannbaum T mit kleinstmöglichem Gewicht, der e nicht enthält, so können wir einen anderen Spannbaum T' konstruieren, der e enthält und ebenfalls das kleinstmögliche Gewicht hat.

Sei $T'' = T \cup \{e\}$. Dann enthält der Graph (V,T'') einen Kreis, auf dem e liegt. Da e zwei verschiedene Komponenten V_i und V_j mit $i \neq j$ verbindet, muss auf diesem Kreis noch eine weitere Kante $e' \in E'$ liegen, die zwei verschiedene Komponenten verbindet. Diese Situation ist in dem folgenden Bild dargestellt:



Aufgrund der Voraussetzung, dass e die billigste Kante aus E' ist, muss $w(e) \leq w(e')$ gelten. Wir betrachten nun die Menge $T' = T'' \setminus \{e'\} = (T \cup \{e\}) \setminus \{e'\}$.

Zunächst können wir festhalten, dass (V, T') zusammenhängend ist. Der Graph (V, T) ist nach Voraussetzung zusammenhängend. Gibt es in (V, T) zwischen zwei Knoten $u, v \in V$ einen Weg, der Kante e' nicht benutzt, so gibt es diesen Weg auch in (V, T'). Gibt es in (V, T) einen Weg P zwischen u und v, der Kante $e' = (w_1, w_2)$ benutzt, so erhält man auch in (V, T') einen Weg zwischen u und v, indem man P nimmt und die Kante e' durch den Weg zwischen w_1 und w_2 über die Kante e ersetzt.

Ebenso können wir argumentieren, dass (V, T') kreisfrei ist: Angenommen es gibt in (V, T') einen Kreis C. Wenn C die Kante e nicht enthält, so gibt es den Kreis C auch in (V, T), im Widerspruch zur Definition von T. Enthält der Kreis die Kante $e = (u_1, u_2)$, so erhalten wir einen Kreis in (V, T), indem wir in C die Kante e durch den Weg zwischen u_1 und u_2 über e' ersetzen.

Somit können wir festhalten, dass (V, T') ein Spannbaum ist. Außerdem gilt

$$w(T') = w(T) + w(e) - w(e') \leq w(T).$$

Damit ist das Lemma bewiesen. □

Wir können nun das folgende Theorem beweisen.

Theorem 4.3.3. *Der Algorithmus von Kruskal berechnet einen minimalen Spannbaum.*

Beweis. Wir benutzen die folgende Invariante, die aus drei Teilen besteht:

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen am Ende des Schleifenrumpfes der for-Schleife stets den Zusammenhangskomponenten des Graphen (V, T), wobei T die in Zeilen 3 und 7 betrachtete Kantenmenge ist.
2. Die Zusammenhangskomponenten von (V, T) stimmen am Ende des Schleifenrumpfes der for-Schleife stets mit den Zusammenhangskomponenten von $(V, \{e_1, \ldots, e_i\})$ überein.
3. Die Kantenmenge $T \subseteq E$ kann am Ende des Schleifenrumpfes der for-Schleife stets zu einer Kantenmenge $T' \supseteq T$ erweitert werden, sodass T' ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Ist diese Invariante korrekt, so ist auch der Algorithmus korrekt. Das sehen wir wie folgt: Am Ende des letzten Schleifendurchlaufes garantiert uns der zweite Teil der Invariante, dass die Zusammenhangskomponenten von (V, T) mit denen von $G = (V, E)$ übereinstimmen. Da G laut Voraussetzung zusammenhängend ist, ist also auch (V, T) zusammenhängend. Ist G keine korrekte Eingabe, da er nicht zusammenhängend ist, wird dies im ersten Schritt des Algorithmus erkannt.

Der dritte Teil der Invariante garantiert uns, dass T zu einer Kantenmenge $T' \supseteq T$ erweitert werden kann, sodass T' ein minimaler Spannbaum von G ist. Da (V, T) bereits zusammenhängend ist, würde die Hinzunahme jeder Kante einen Kreis schließen. Somit muss $T' = T$ gelten, woraus direkt folgt, dass $T = T'$ ein minimaler Spannbaum von G ist.

Nun müssen wir nur noch die Korrektheit der Invariante zeigen. Dazu überlegen wir uns zunächst, dass die Invariante nach dem ersten Schleifendurchlauf gilt. In diesem ersten

Durchlauf fügen wir die billigste Kante e_1 in die bisher leere Menge T ein, da ihre beiden Endknoten u_1 und v_1 zu Beginn in verschiedenen einelementigen Mengen liegen. Nach dem Einfügen von e_1 vereinigen wir diese beiden einelementigen Mengen, so dass nach dem ersten Schleifendurchlauf die Knoten u_1 und v_1 in derselben und alle anderen Knoten in getrennten einelementigen Mengen liegen. Dies sind genau die Zusammenhangskomponenten von $(V, T) = (V, \{e_1\})$. Damit sind der erste und zweite Teil der Invariante gezeigt. Für den dritten Teil wenden wir Lemma 4.3.2 an. Wählen wir in diesem Lemma $V_1 = \{v_1\}$,...,$V_m = \{v_m\}$, wobei wir $V = \{v_1, \ldots, v_m\}$ annehmen, so impliziert es direkt, dass wir die Menge $T = \{e_1\}$ zu einem minimalen Spannbaum von G erweitern können.

Schauen wir uns nun einen Schleifendurchlauf an und nehmen an, dass die Invariante am Ende des vorherigen Schleifendurchlaufs erfüllt war.

1. Falls wir Kante e_i nicht in T einfügen, so ändern sich weder die Mengen in der Union-Find-Datenstruktur noch die Zusammenhangskomponenten von (V, T). Teil eins der Invariante bleibt dann also erhalten. Fügen wir e_i in T ein, so fallen die Zusammenhangskomponenten von u_i und v_i in (V, T) zu einer gemeinsamen Komponente zusammen. Diese Veränderung bilden wir in der Union-Find-Datenstruktur korrekt ab. Also bleibt auch dann der erste Teil erhalten.

2. Bezeichne $V_1, \ldots, V_k$ die Zusammenhangskomponenten von (V, T) am Ende des letzten Schleifendurchlaufs und bezeichne $T_1, \ldots, T_k$ die Kanten innerhalb dieser Komponenten. Aufgrund der Invariante können wir $T = T_1 \cup \ldots \cup T_k$ zu einem minimalen Spannbaum von $G = (V, E)$ erweitern. Fügen wir in dieser Iteration e_i zu T hinzu, so garantiert Lemma 4.3.2, dass wir auch $T \cup \{e\}$ zu einem minimalen Spannbaum erweitern können: Da alle Kanten $e_1, \ldots, e_{i-1}$ aufgrund des zweiten Teils der Invariante innerhalb der Komponenten $V_1, \ldots, V_k$ verlaufen, ist e_i die billigste Kante, die zwei verschiedene Komponenten miteinander verbindet. Somit sind die Voraussetzungen von Lemma 4.3.2 erfüllt.

3. Falls wir e_i nicht zu T hinzufügen, so verläuft es innerhalb einer der gerade definierten Komponenten $V_1, \ldots, V_k$. Das heißt die Komponenten von $(V, \{e_1, \ldots, e_{i-1}\})$ und $(V, \{e_1, \ldots, e_i\})$ sind identisch und die Invariante bleibt erhalten. Fügen wir e_i zu T hinzu, so verschmelzen in $(V, \{e_1, \ldots, e_i\})$ die Komponenten, die u_i und v_i enthalten. Diese Verschmelzung passiert aber genauso in (V, T), da wir e_i zu T hinzufügen. □

Laufzeit

Nachdem wir die Korrektheit des Algorithmus von Kruskal bewiesen haben, analysieren wir nun noch die Laufzeit. Dafür ist es zunächst wichtig, wie effizient wir die Operationen der Union-Find-Datenstruktur implementieren können. Wir betrachten dazu die folgende Realisierung der Union-Find-Datenstruktur, die immer dann benutzt werden kann, wenn am Anfang (so wie im Algorithmus von Kruskal) feststeht, wie oft MAKE-SET aufgerufen wird.

Bezeichne n die Anzahl von Aufrufen von MAKE-SET, die wir vorher als Parameter übergeben bekommen. Wir legen ein Array A der Länge n an und speichern für jedes der n Objekte den Repräsentanten der Menge, in der es sich gerade befindet. Am Anfang initialisieren wir das Array einfach mit $A[1] = 1, \ldots, A[n] = n$.

Mit dieser Information allein könnten wir bereits eine Union-Find-Datenstruktur implementieren. Allerdings würde jede Vereinigung Zeit $\Theta(n)$ benötigen, da wir bei einer Vereinigung

zweier Mengen jedes Element darauf testen müssten, ob es zu einer dieser Mengen gehört. Deshalb speichern wird noch weitere Informationen.

Wir legen ein Array L der Länge n an und speichern in jedem der Einträge einen Zeiger auf eine verkettete Liste. Wenn ein Element i Repräsentant seiner Menge ist, so enthält die verkettete Liste $L[i]$ alle Objekte in der Menge, die i repräsentiert. Zusätzlich speichern wir in einem weiteren Array size die Kardinalität der Listen von L ab. Ist ein Objekt i nicht Repräsentant seiner Menge, so gelte $L[i] - >$ **null** und size$[i] = 0$. Zu Beginn repräsentieren alle Objekte ihre eigenen einelementigen Teilmengen und wir setzen $L[i] \to i \to$ **null** und size$[i] = 1$.

Die n Aufrufe von MAKE-SET haben wir mit dieser Initialisierung bereits erledigt. Diese benötigen insgesamt eine Laufzeit von $O(n)$. Wir überlegen uns nun, wie man UNION und FIND implementiert. FIND(x) ist sehr einfach; wir geben lediglich $L[x]$ zurück. Dafür benötigen wir nur konstante Laufzeit $O(1)$.

Die UNION-Operation sieht wie folgt aus:

```
    UNION(x, y)
1   i = FIND(x); j = FIND(y);
2   if size[i] > size[j] then {Vertausche x und y und vertausche i und j.}
3   for all z ∈ L[i] { A[z] = j }
4   Hänge L[i] and L[j] an und setze L[i] → null und size[i] = 0.
5   size[j] = size[i] + size[j]
```

Der Repräsentant der vereinigten Menge ist also der alte Repräsentant der größeren Menge und die Einträge im Array A werden für alle Elemente der kleineren Menge entsprechend angepasst. Die Laufzeit ist somit $O(\min\{\text{size}[i], \text{size}[j]\})$, wobei i und j die Repräsentanten der beiden zu vereinigenden Mengen bezeichnen.

Lemma 4.3.4. *Jede Folge von* $(n-1)$ UNION- *und* f FIND-*Operationen kann in Zeit* $O(n \log(n) + f)$ *durchgeführt werden.*

Beweis. Da FIND-Operationen in Zeit $O(1)$ ausgeführt werden können, ist nur zu zeigen, dass alle UNION-Operationen zusammen eine Laufzeit von $O(n \log n)$ haben. Wir nehmen an, dass jede UNION-Operation höchstens c mal so viele Rechenschritte benötigt wie die Kardinalität der kleineren Menge, für eine geeignete Konstante $c > 0$. Bezeichne $A_1, \ldots, A_{n-1}$ die kleineren Mengen, die bei den $n-1$ UNION-Operationen auftreten. Dann können wir die Laufzeit abschätzen durch

$$c \cdot \sum_{i=1}^{n-1} \sum_{x \in A_i} 1 = c \cdot \sum_{j=1}^{n} |\{A_i \mid j \in A_i\}|. \tag{4.1}$$

Wir schauen also für jedes Objekt j, bei wie vielen UNION-Operationen es in der kleineren Menge auftritt und summieren diese Anzahlen auf.

Tritt ein Element j zum ersten Mal in der kleineren Menge auf, so hat diese Kardinalität 1. Danach liegt j in einer Menge der Größe mindestens 2. Ist diese Menge dann später wieder die kleinere Menge einer UNION-Operation, so liegt j hinterher in einer Menge der Größe mindestens 4 usw. Wir können festhalten, dass das Element j, nachdem es s mal in der kleineren Menge einer UNION-Operation lag, in einer Menge der Größe mindestens 2^s liegt.

Da es nur n Elemente gibt, muss $2^s \leq n$ und somit $s \leq \log_2 n$ gelten. Also gilt für jedes Element j

$$|\{A_i \mid j \in A_i\}| \leq \log_2 n.$$

Zusammen mit (4.1) ergibt sich als obere Abschätzung für die Anzahl an Operationen aller UNION-Operationen

$$c \cdot \sum_{j=1}^{n} \log_2 n = O(n \log n). \qquad \square$$

Nun können wir die Laufzeit des Algorithmus von Kruskal abschließend bestimmen.

Theorem 4.3.5. *Für zusammenhängende ungerichtete Graphen mit m Kanten benötigt der Algorithmus von Kruskal Laufzeit $O(m \log m) = O(|E| \cdot \log |V|)$.*

Beweis. Die Tiefensuche im ersten Schritt benötigt Zeit $O(|V| + m)$. Da der Graph laut Voraussetzung zusammenhängend ist, gilt $m \geq |V| - 1$, also $O(|V| + m) = O(m)$. Das Sortieren der Kanten benötigt Zeit $O(m \log m)$. Die for-Schleife wird m mal durchlaufen und abgesehen von UNION und FIND werden in jedem Durchlauf nur konstant viele Operationen durchgeführt, so dass wir eine Laufzeit von $O(m)$ für alle Operationen in der Schleife abgesehen von UNION und FIND ansetzen können.

Wir führen in der Union-Find-Datenstruktur $2m$ FIND-Operationen und $m - 1$ UNION-Operationen durch. Mit Lemma 4.3.4 ergibt sich also eine Laufzeit von $O(m \log m + 2m) = O(m \log m)$. Wegen $|E| \leq |V|^2$ folgt $O(m \log m) = O(|E| \cdot \log |V|^2) = O(|E| \cdot \log |V|)$ Damit ist das Theorem bewiesen. $\square$

Literatur

Der Algorithmus von Kruskal kann in Kapitel 23 von [1] nachgelesen werden. Kapitel 21 enthält zudem eine Beschreibung von Union-Find-Datenstrukturen.

4.4 Kürzeste Wege

Sei $G = (V, E)$ ein gerichteter Graph und sei $w : E \to \mathbb{R}$ eine Gewichtung der Kanten. Da wir uns in diesem Abschnitt mit kürzesten Wegen beschäftigen, werden wir $w(e)$ auch als die *Länge der Kante e* bezeichnen.

Definition 4.4.1. *Für zwei gegebene Knoten $s, t \in V$ ist $P = (v_0, v_1, \ldots, v_\ell)$ ein* Weg von s nach t*, wenn $v_0 = s$, $v_\ell = t$ und wenn $(v_i, v_{i+1}) \in E$ für alle $i \in \{0, 1, \ldots, \ell - 1\}$ gilt. Wir definieren die* Länge von P *als $w(P) = \sum_{i=0}^{\ell-1} w(v_i, v_{i+1})$. Wir sagen, dass P ein* kürzester Weg von s nach t *ist, falls es keinen anderen Weg P' von s nach t mit $w(P') < w(P)$ gibt. Wir nennen die Länge $w(P)$ des kürzesten Weges P die* Entfernung von s nach t *und bezeichnen diese mit $\delta(s, t)$. Existiert kein Weg von s nach t, so gelte $\delta(s, t) = \infty$.*

Zur besseren Intuition können wir uns im Folgenden vorstellen, dass die Knoten des Graphen Kreuzungen entsprechen und dass für jede Straße von Kreuzung $u \in V$ zu Kreuzung $v \in V$ eine Kante $e = (u, v)$ in E enthalten ist, wobei $w(e) = w((u, v))$ die Länge der Straße angibt.

Es gibt jedoch auch zahlreiche andere Anwendungskontexte, in denen kürzeste Wege gefunden werden müssen. Insbesondere auch solche, in denen negative Kantengewichte auftreten können.

Wir sind an der Lösung der folgenden zwei Probleme interessiert:

1. Im *Single-Source Shortest Path Problem (SSSP)* ist zusätzlich zu G und w ein Knoten $s \in V$ gegeben und wir möchten für jeden Knoten $v \in V$ einen kürzesten Weg von s nach v und die Entfernung von s nach v berechnen.

2. Im *All-Pairs Shortest Path Problem (APSP)* sind nur G und w gegeben und wir möchten für jedes Paar $u, v \in V$ einen kürzesten Weg von u nach v und die Entfernung von u nach v berechnen.

Für den Spezialfall, dass $w(e) = 1$ für alle Kanten $e \in E$ gilt, haben wir mit Breitensuche bereits einen Algorithmus kennengelernt, der das SSSP löst. Breitensuche lässt sich allerdings nicht direkt auf den Fall verallgemeinern, dass wir beliebige Kantengewichte haben.

Bevor wir Algorithmen für das SSSP und das APSP beschreiben, schauen wir uns zunächst einige strukturelle Eigenschaften von kürzesten Wegen an.

Optimale Substruktur

Für die Algorithmen, die wir vorstellen werden, ist es wichtig, dass ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$ aus kürzesten Wegen zwischen anderen Knoten zusammengesetzt ist.

Lemma 4.4.2. *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Für jedes Paar i, j mit $0 \leq i < j \leq \ell$ ist $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ ein kürzester Weg von v_i nach v_j.*

Beweis. Wir zerlegen den Weg $P = P_{0\ell}$ in die drei Teilwege P_{0i}, P_{ij} und $P_{j\ell}$. Dies schreiben wir als

$$P = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

Insbesondere gilt $w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{j\ell})$.

Nehmen wir nun an, dass P_{ij} kein kürzester Weg von v_i nach v_j ist. Dann gibt es einen Weg P'_{ij} von i nach j mit $w(P'_{ij}) < w(P_{ij})$. Dann erhalten wir ebenfalls einen anderen Weg P' von v_0 nach v_ℓ:

$$P' = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P'_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

Für diesen Weg gilt $w(P') = w(P) - w(P_{ij}) + w(P'_{ij}) < w(P)$ im Widerspruch zur Voraussetzung, dass P ein kürzester Weg von v_0 nach v_ℓ ist. Also kann es keinen kürzeren Weg als P_{ij} von v_i nach v_j geben. □

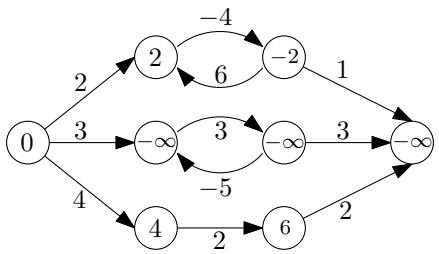
Daraus erhalten wir direkt das folgende Korollar.

Korollar 4.4.3. *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Dann gilt*

$$\delta(v_0, v_\ell) = \delta(v_0, v_{\ell-1}) + w(v_{\ell-1}, v_\ell).$$

Negative Kantengewichte

Wir haben in der obigen Definition auch erlaubt, dass Kantengewichte negativ sein können. Dies kann in einigen Anwendungskontexten tatsächlich auftreten, es führt jedoch zu zusätzlichen Schwierigkeiten, insbesondere dann, wenn der Graph einen Kreis mit negativem Gesamtgewicht enthält. Dann ist nämlich für einige Paare von Knoten kein kürzester Weg mehr definiert, da man beliebig oft den Kreis mit negativem Gesamtgewicht entlang laufen könnte. Die folgende Abbildung zeigt einen Graphen mit einem negativen Kreis. Die Zahlen in den Knoten entsprechen dabei den Entfernungen der Knoten vom linken Knoten.



Der *Algorithmus von Dijkstra*, den wir für das SSSP kennenlernen werden, funktioniert nur für Graphen, die keine negativen Kantengewichte haben. Der *Floyd-Warshall-Algorithmus* für APSP funktioniert auch für negative Kantengewichte solange es keine negativen Kreise gibt. Gibt es einen negativen Kreis, so kann der Floyd-Warshall-Algorithmus zumindest benutzt werden, die Existenz eines solchen zu verifizieren.

4.4.1 Single-Source Shortest Path Problem

Es sei ein Graph $G = (V, E)$, eine Gewichtung $w : E \to \mathbb{R}_{\geq 0}$ und ein Startknoten $s \in V$ gegeben. In diesem Abschnitt gehen wir davon aus, dass alle Kantengewichte nicht-negativ sind.

Darstellung kürzester Wege

Zunächst halten wir fest, dass ein kürzester Weg in einem Graphen ohne negativen Kreis keinen Knoten mehrfach besuchen muss. Wäre $P = (v_0, \ldots, v_\ell)$ nämlich ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$ und gäbe es v_i und v_j mit $i < j$ und $v_i = v_j$, so wäre der Teilweg $P_{ij} = (v_i, \ldots, v_j)$ ein Kreis, der laut Voraussetzung keine negativen Gesamtkosten hat. Das heißt, er hat entweder Kosten 0 oder echt positive Kosten. In jedem Falle können wir einen neuen Weg P' von v_0 nach v_ℓ konstruieren, indem wir den Kreis überspringen: $P' = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_\ell)$. Für diesen neuen Weg P' gilt $w(P') \leq w(P)$. Somit können wir davon ausgehen, dass die kürzesten Wege zwischen zwei Knoten, die wir berechnen werden, aus höchsten $n - 1$ Kanten bestehen.

Beim SSSP berechnen wir n kürzeste Wege, nämlich einen vom Startknoten s zu jedem anderen Knoten. Beim APSP berechnen wir sogar $\binom{n}{2}$ kürzeste Wege. Da wir als obere Abschätzung für die Anzahl Kanten auf einem kürzesten Weg nur $n - 1$ haben, müssten wir damit rechnen, dass wir Platz $\Theta(n^2)$ bzw. $\Theta(n^3)$ brauchen, um alleine die Ausgabe aufzuschreiben. Wir wollen uns nun überlegen, wie man die Ausgabe effizienter kodieren

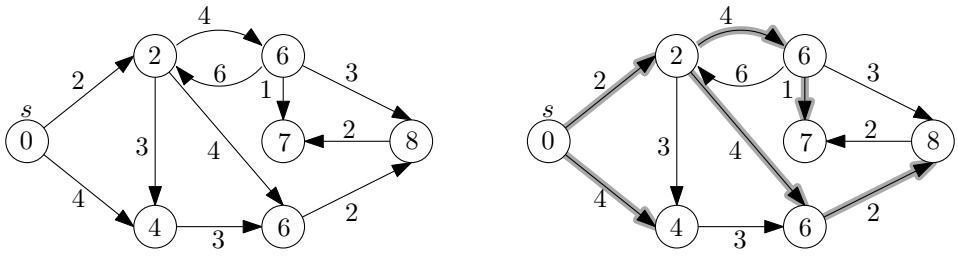
kann. Dabei machen wir uns die Eigenschaft der optimalen Substruktur aus Lemma 4.4.2 zu Nutze.

Wir betrachten in diesem Abschnitt nur das SSSP. Statt alle n Wege separat zu speichern, wird der Algorithmus von Dijkstra einen Kürzeste-Wege-Baum mit Wurzel s gemäß der folgenden Definition berechnen.

Definition 4.4.4. *Wir nennen (V', E') mit $V' \subseteq V$ und $E' \subseteq E$ einen* Kürzeste-Wege-Baum mit Wurzel s*, wenn die folgenden Eigenschaften erfüllt sind.*

1. *V' ist die Menge der Knoten, die von s aus in G erreichbar sind.*
2. *G' ist ein gewurzelter Baum mit Wurzel s. (Das bedeutet, dass G' azyklisch ist, selbst wenn wir die Richtung der Kanten ignorieren, und dass alle Kanten von s weg zeigen.)*
3. *Für alle $v \in V'$ ist der eindeutige Weg von s nach v in G' ein kürzester Weg von s nach v in G.*

Kürzeste-Wege-Bäume benötigen Platz $O(n)$, da wir für jeden Knoten nur seinen Vorgänger speichern müssen. Sie sind also eine platzsparende Möglichkeit, um alle kürzesten Wege von s zu den anderen Knoten des Graphen zu kodieren. Die folgende Abbildung zeigt rechts einen Kürzeste-Wege-Baum für den Graphen auf der linken Seite.



Relaxation von Kanten

Der Algorithmus von Dijkstra verwaltet für jeden Knoten $v \in V$ zwei Attribute: $\pi(v) \in V \cup \{\textbf{null}\}$ und $d(v) \in \mathbb{R}$. Der Knoten $\pi(v)$ wird am Ende des Algorithmus der Vorgänger von v in einem Kürzeste-Wege-Baum mit Wurzel s sein (sofern $\pi(v) \neq \textbf{null}$) und die Zahl $d(v) \in \mathbb{R}$ wird am Ende die Entfernung von s nach v angeben. Wir initialisieren $\pi(v)$ und $d(v)$ wie folgt:

```
    INITIALIZE-DIJKSTRA(G, s)
1   for each v ∈ V {
2       d(v) = ∞
3       π(v) = null
4   }
5   d(s) = 0
```

Im Algorithmus wird $d(v) \geq \delta(s, v)$ zu jedem Zeitpunkt und für jeden Knoten $v \in V$ gelten. Das heißt, $d(v)$ ist stets eine obere Schranke für die Länge des kürzesten Weges von s nach v. Direkt nach der Initialisierung ist dies sicherlich der Fall. Die Idee ist nun, die oberen Schranken Schritt für Schritt zu verbessern, bis irgendwann $d(v) = \delta(s, v)$ für alle Knoten v gilt. Dazu benutzen wir die folgende Beobachtung.

Beobachtung 4.4.5. *Für $x, y, z \in V$ gilt stets*

$$\delta(x, y) \leq \delta(x, z) + w(z, y).$$

Beweis. Wir konstruieren einen Weg von x nach y, indem wir von x auf einem kürzesten Weg nach z laufen und dann der direkten Kanten von z zu y folgen. Dieser Pfad hat Länge $\delta(x, z) + w(z, y)$. Da er offensichtlich nicht kürzer sein kann als der kürzeste Weg von x nach y, muss $\delta(x, z) + w(z, y) \geq \delta(x, y)$ gelten. □

Aus dieser Beobachtung folgt direkt eine Möglichkeit, neue obere Schranken zu erhalten: Seien $u, v \in V$ und gelte $d(u) \geq \delta(s, u)$. Dann gilt

$$\delta(s, v) \leq d(u) + w(u, v).$$

Das nutzen wir, um die oberen Schranken mittels der folgenden Methode zu verbessern.

```
    RELAX(u, v)
1   if (d(v) > d(u) + w(u, v)) {
2       d(v) = d(u) + w(u, v)
3       π(v) = u
4   }
```

Aus den bisherigen Überlegungen folgt direkt das folgende Lemma.

Lemma 4.4.6. *Nach einem Aufruf von* INITIALIZE-DIJKSTRA *gefolgt von einer beliebigen Sequenz von* RELAX*-Aufrufen gilt $d(v) \geq \delta(s, v)$ für alle Knoten v. Das heißt, der Wert $d(v)$ ist zu jedem Zeitpunkt eine obere Schranke für die Entfernung $\delta(s, v)$ von s nach v.*

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra wird eine Folge von RELAX-Aufrufen durchführen und sicherstellen, dass am Ende tatsächlich $d(v) = \delta(s, v)$ für alle Knoten v gilt. In der folgenden Implementierung wird $S \subseteq V$ eine Menge von Knoten bezeichnen, sodass für jeden Knoten $v \in S$ bereits $d(v) = \delta(s, v)$ gilt. Es sei außerdem $Q = V \setminus S$. Wir gehen davon aus, dass der Graph G in Adjazenzlistendarstellung gegeben ist. Die Methode EXTRACT-MIN(Q) entfernt einen Knoten $u \in Q$ aus Q, der unter all diesen Knoten den kleinsten Wert $d(u)$ hat, und gibt diesen Knoten zurück.

```
    DIJKSTRA(G, w, s)
1   INITIALIZE-DIJKSTRA(G, s)
2   S = ∅; Q = V;
3   while Q ≠ ∅ {
4       u = EXTRACT-MIN(Q)
5       S = S ∪ {u}
6       for each v ∈ Adj[u] {
7           RELAX(u, v)
8       }
9   }
```

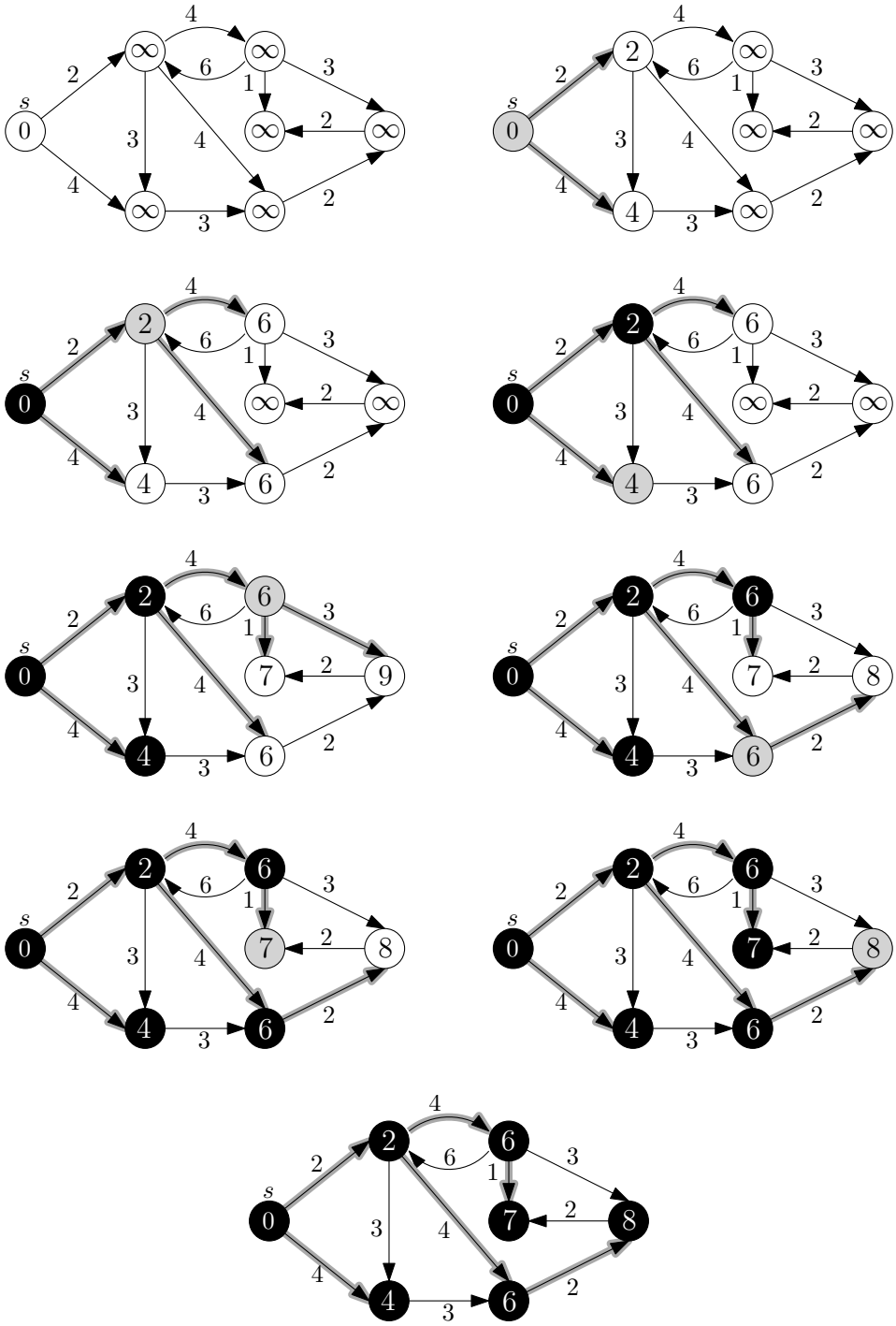


Abbildung 4.1: Beispiel für den Ablauf des Dijkstra-Algorithmus. Schwarze Knoten gehören zu S und der graue Knoten ist jeweils der Knoten u, der aus Q extrahiert wird. Eine graue Kanten (u, v) bedeutet, dass $\pi(v) = u$ gilt.

Ein Beispiel für die Ausführung des Algorithmus von Dijkstra findet sich in Abbildung 4.1.

Theorem 4.4.7. *Der Algorithmus von Dijkstra terminiert auf gerichteten Graphen $G = (V, E)$ mit nicht-negativen Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$ und Startknoten $s \in V$ in einem Zustand, in dem $d(v) = \delta(s, v)$ für alle $v \in V$ gilt.*

Beweis. Wir beweisen das Theorem mit Hilfe der folgenden Invariante, bei der wir die Konvention $w((u, v)) = \infty$ für $(u, v) \notin E$ anwenden.

> Am Ende jeder Ausführung der **while**-Schleife in Zeilen 4–8 gilt:
>
> 1. $\forall v \in S : d(v) = \delta(s, v)$,
> 2. $\forall v \in Q = V \setminus S : d(v) = \min\{\delta(s, x) + w(x, v) \mid x \in S\}$.

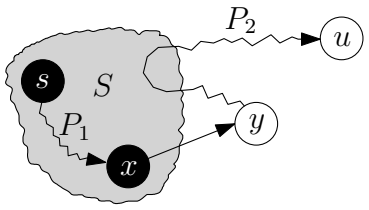
Gilt diese Invariante, so folgt aus ihr direkt das Theorem: In jedem Schritt der **while**-Schleife fügen wir einen weiteren Knoten aus Q zu der Menge S hinzu. Da zu jedem Zeitpunkt $S \cup Q = V$ gilt, muss nach n Durchläufen der **while**-Schleife $S = V$ gelten. Dann ist $Q = \emptyset$ und der Algorithmus terminiert. Die Invariante besagt dann, dass $d(v) = \delta(s, v)$ für alle Knoten $v \in S = V$ gilt.

Am Ende des ersten Schleifendurchlaufs gilt $S = \{s\}$ und $d(s) = \delta(s, s) = 0$. Somit ist der erste Teil der Invariante erfüllt. Den zweiten Teil der Invariante können wir in diesem Falle umschreiben zu: $\forall v \in V \setminus \{s\} : d(v) = \delta(s, s) + w(s, v) = w(s, v)$. Die Gültigkeit dieser Aussage folgt daraus, dass wir für jeden Knoten $v \in \text{Adj}[s]$ die Methode Relax(s, v) aufgerufen haben, die laut ihrer Definition $d(v) = \min\{d(s) + w(s, v), \infty\} = w(s, v)$ gesetzt hat.

Wir müssen uns nun nur noch überlegen, dass die Invariante erhalten bleibt. Gehen wir also davon aus, wir befinden uns in einem Durchlauf der **while**-Schleife und am Ende des vorherigen Durchlaufs galt die Invariante. Während dieser Iteration fügen wir den Knoten $u \in Q$, der momentan unter allen Knoten aus Q den kleinsten Wert $d(u)$ hat, zur Menge S hinzu.

Wir zeigen zunächst den ersten Teil der Invariante, indem wir zeigen, dass für diesen Knoten $d(u) = \delta(s, u)$ gilt. Sei dazu P ein kürzester Weg von s nach u und sei y der erste Knoten auf dem Pfad P, der nicht zur Menge S gehört. Da u selbst noch nicht zu S gehört, muss es einen solchen Knoten y geben. Weiterhin muss y einen Vorgänger haben, da der erste Knoten auf dem Weg P der Knoten $s \in S$ ist. Wir nennen diesen Vorgänger x. Den Teilweg von P von s nach x nennen wir P_1 und den Teilweg von y nach u nennen wir P_2. Diese Teilwege können auch leer sein.

Die Notation ist in der folgenden Abbildung veranschaulicht.



Wegen Lemma 4.4.2 muss P_1 ein kürzester Weg von s nach x sein. Somit gilt

$$\delta(s,u) = w(P_1) + w(x,y) + w(P_2) = \delta(s,x) + w(x,y) + w(P_2) \geq d(y) + w(P_2) \geq d(y),$$

wobei wir bei der vorletzten Ungleichung den zweiten Teil der Invariante für $y \in Q$ ausgenutzt haben. Wegen Lemma 4.4.6 gilt $d(u) \geq \delta(s,u)$ und somit folgt insgesamt $d(u) \geq d(y)$. Da u ein Knoten aus Q mit kleinstem d-Wert ist, muss auch gelten $d(u) \leq d(y)$. Alles in allem erhalten wir die Ungleichungskette

$$d(u) \geq \delta(s,u) \geq d(y) \geq d(u),$$

deren einzige Lösung $d(u) = \delta(s,u)$ ist. Dies ist genau die Aussage, die wir für den ersten Teil der Invariante zeigen mussten.

Der zweite Teil der Invariante folgt nun einfach daraus, dass vorher $d(v) = \min\{\delta(s,x) + w(x,v) \mid x \in S\}$ für all $v \in Q$ galt und dass nun für jeden Knoten aus $v \in \text{Adj}[u]$ die Operation Relax(u,v) aufgerufen wird, die dazu führt, dass gilt:

$$\begin{aligned} d(v) &= \min\{\min\{\delta(s,x) + w(x,v) \mid x \in S\}, \delta(s,u) + w(u,v)\} \\ &= \min\{\delta(s,x) + w(x,v) \mid x \in S \cup \{u\}\}. \end{aligned}$$ □

Nun fehlt nur noch der Beweis, dass die Vorgänger-Relation, die in π berechnet wird, wirklich einen Kürzesten-Wege-Baum mit Wurzel s ergibt. Dies folgt unmittelbar aus dem folgenden Lemma, welches wir in der Vorlesung jedoch nicht beweisen werden.

Lemma 4.4.8. *Es gelte nach einem Aufruf von* Initialize-Dijkstra *gefolgt von einer Sequenz von* Relax*-Aufrufen $d(v) = \delta(s,v)$ für jeden Knoten v. Sei*

$$V_\pi = \{v \in V \mid \pi(v) \neq \textbf{null}\} \cup \{s\} \quad \textit{und} \quad E_\pi = \{(\pi(v), v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

Dann ist der Graph (V_π, E_π) ein Kürzeste-Wege-Baum mit Wurzel s.

Implementierung und Laufzeit Wir werden nun die Laufzeit des Algorithmus von Dijkstra analysieren. Dazu müssen wir uns zunächst überlegen, in welcher Datenstruktur wir die Menge Q verwalten. Diese Datenstruktur muss eine Menge verwalten können und sollte die folgenden drei Operationen möglichst effizient unterstützen:

- Insert(Q,x,d): Füge ein neues Element x mit Schlüssel $d \in \mathbb{R}$ in die Menge Q ein.
- Extract-Min(Q): Entferne aus Q ein Element mit dem kleinsten Schlüssel und gib dieses Element zurück.
- Change-Key(Q,x,d_1,d_2): Ändere den Schlüssel des Objektes $x \in Q$ von d_1 auf d_2.

Eine solche Datenstruktur heißt *Min-Priority Queue* und sie lässt sich leicht durch z. B. eine verkettete Liste realisieren, wenn wir auf Effizienz keinen Wert legen. Sei n die Anzahl der Einträge der Priority Queue, dann würde eine Realisierung als verkettete Liste Zeit $\Theta(n)$ für Extract-Min und Decrease-Key benötigen. Dies wollen wir nun verbessern.

Wir haben mit AVL-Bäumen schon eine Möglichkeit in der Vorlesung kennengelernt, um Priority Queues effizienter zu realisieren. Wir legen dafür einen AVL-Baum an und fügen bei Insert(Q,x,d) einfach das Objekt x mit Schlüssel d in den AVL-Baum ein. Wir müssen

auf den Fall aufpassen, dass wir zwei Objekte x und y mit dem gleichen Schlüssel einfügen wollen. Wir nehmen für diesen Fall an, dass es eine inhärente Ordnung auf den Objekten gibt und dass bei eigentlich gleichem Schlüssel d diese inhärente Ordnung entscheidet, welcher Schlüssel als kleiner betrachtet wird. In unserem Beispiel sind die Objekte in Q die Knoten aus V. Wir können davon ausgehen, dass diese mit $v_1, \ldots, v_n$ durchnummeriert sind und falls wir v_i und v_j mit gleichem Schlüssel d in Q einfügen, so wird v_i als kleiner als v_j betrachtet, wenn $i < j$ gilt. Auf diese Weise ist stets eine eindeutige Ordnung auf den Schlüsseln gegeben. Da die Höhe eines AVL-Baumes mit n Knoten durch $O(\log(n))$ beschränkt ist, können wir INSERT(Q, x, d) in Zeit $O(\log(n))$ ausführen.

EXTRACT-MIN(Q) können wir in einem AVL-Baum realisieren, indem wir ausgehend von der Wurzel immer dem Zeiger zum linken Kind folgen. Auf diese Weise finden wir das Objekt mit dem kleinsten Schlüssel. Auch dies geht wegen der beschränkten Höhe in Zeit $O(\log(n))$. Haben wir es einmal gefunden, so können wir es in Zeit $O(1)$ löschen und zurückgeben. Ebenso können wir DECREASE-KEY in Zeit $O(\log(n))$ realisieren, indem wir zunächst nach x suchen (dafür ist wichtig, dass wir seinen momentanen Schlüssel d_1 übergeben bekommen), es dann löschen und mit dem neuen Schlüssel d_2 wieder einfügen.

Theorem 4.4.9. *Die Laufzeit des Algorithmus von Dijkstra beträgt $O((n+m)\log n)$.*

Beweis. Die Initialisierung erfolgt in Zeit $O(n)$. Da die Adjazenzliste eines Knoten $v \in V$ dann das einziges Mal durchlaufen wird, wenn der Knoten v der Menge S hinzugefügt wird, wird für jede Kante $e = (u, v) \in E$ genau einmal die Operation RELAX(u, v) aufgerufen. Innerhalb des Aufrufs von RELAX(u, v) wird gegebenenfalls der Wert $d(v)$ reduziert. Dies entspricht einem Aufruf von CHANGE-KEY, welcher Laufzeit $O(\log n)$ benötigt. Alle Aufrufe der RELAX-Methode zusammen haben somit eine Laufzeit von $O(m \log n)$. Wir benötigen außerdem genau n Aufrufe von EXTRACT-MIN. Diese haben eine Gesamtlaufzeit von $O(n \log n)$. Damit folgt das Theorem. □

4.4.2 All-Pairs Shortest Path Problem

In diesem Abschnitt beschäftigen wir uns mit dem APSP. Dazu sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$ gegeben. Im Gegensatz zum Dijkstra-Algorithmus funktioniert der Floyd-Warshall-Algorithmus, den wir gleich beschreiben werden, auch, wenn es negative Kantengewichte gibt. Enthält der Graph einen negativen Kreis, so stellt der Algorithmus dies fest und bricht ab.

Für den Floyd-Warshall-Algorithmus gehen wir davon aus, dass $V = \{1, \ldots, n\}$ gilt, und wir untersuchen Wege, die nur bestimmte Knoten benutzen dürfen. Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von v_0 nach v_ℓ. Wir haben bereits im SSSP argumentiert, dass es für jedes Paar von Knoten u und v in einem Graphen ohne negativen Kreis stets einen kürzesten Weg von u nach v gibt, der keinen Knoten mehrfach besucht. Wir nehmen an, dass dies auf P zutrifft, das heißt, die Knoten $v_0, \ldots, v_\ell$ seien paarweise verschieden. Wir nennen einen solchen Weg auch *einfachen Weg*. Dann nennen wir $v_1, \ldots, v_{\ell-1}$ die *Zwischenknoten von P*. Wir schränken nun die Menge der erlaubten Zwischenknoten ein.

Wir betrachten für jedes Paar $i, j \in V$ alle einfachen Wege von i nach j, die nur Zwischenknoten aus der Menge $\{1, \ldots, k\}$ für ein bestimmtes k benutzen dürfen. Sei P_{ij}^k der kürzeste solche Weg von i nach j und sei $\delta_{ij}^{(k)}$ seine Länge. Wir überlegen uns, wie wir den Weg P_{ij}^k und $\delta_{ij}^{(k)}$ bestimmen können, wenn wir für jedes Paar $i, j \in V$ bereits den Weg P_{ij}^{k-1} kennen,

also den kürzesten Weg von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k-1\}$ benutzen darf. Hierzu unterscheiden wir zwei Fälle:

- Enthält der Weg P_{ij}^k den Knoten k nicht als Zwischenknoten, so enthält er nur die Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten und somit ist P_{ij}^k auch dann ein kürzester Weg, wenn wir nur diese Knoten als Zwischenknoten erlauben. Es gilt dann also $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$ und $P_{ij}^k = P_{ij}^{k-1}$.
- Enthält der Weg P_{ij}^k den Knoten k als Zwischenknoten, so zerlegen wir ihn wie folgt in zwei Teile:
$$P_{ij}^k = i \overset{P}{\rightsquigarrow} k \overset{P'}{\rightsquigarrow} j.$$
Da P_{ij}^k ein einfacher Weg ist, kommt k nur einmal vor und ist somit nicht in den Teilwegen P und P' enthalten. Diese müssen laut Lemma 4.4.2 kürzeste Wege von i nach k bzw. von k nach j sein, die nur Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten benutzen. Somit gilt in diesem Fall
$$\delta_{ij}^{(k)} = \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}.$$
und
$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j.$$

Aus dieser Beobachtung können wir direkt rekursive Formeln zur Berechnung der Entfernungen $\delta^{(k)}(i,j)$ und der Wege P_{ij}^k herleiten. Wir betrachten zunächst nur die Entfernungen. Für alle $i, j \in V$ gilt

$$\delta_{ij}^{(k)} = \begin{cases} w(i,j) & \text{für } k = 0 \\ \min\{\delta_{ij}^{(k-1)}, \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}\} & \text{für } k > 0. \end{cases}$$

Für $k = n$ erlauben wir alle Knoten $\{1, \ldots, n\}$ als Zwischenknoten. Demzufolge besteht für $n = k$ keine Einschränkung mehr und es gilt $\delta(i,j) = \delta_{ij}^{(n)}$ für alle $i, j \in V$.

Wir weisen an dieser Stelle nochmals explizit darauf hin, dass wir für diese Rekursionsformel essentiell ausgenutzt haben, dass es im Graphen G keinen Kreis mit negativem Gesamtgewicht gibt. Gibt es nämlich einen solchen Kreis, der von einem Knoten $i \in V$ erreichbar ist und von dem aus der Knoten $j \in V$ erreichbar ist, so ist der kürzeste Weg von i nach j nicht mehr einfach. Er würde stattdessen den Kreis unendlich oft durchlaufen und demzufolge die Knoten auf dem Kreis mehrfach besuchen.

Floyd-Warshall-Algorithmus

Der Floyd-Warshall-Algorithmus nutzt die rekursive Darstellung der $\delta_{ij}^{(k)}$, die wir soeben hergeleitet haben, um das APSP zu lösen. Zur Vereinfachung der Notation gehen wir davon aus, dass $V = \{1, \ldots, n\}$ gilt und dass der Graph als $(n \times n)$-Adjazenzmatrix $W = (w_{ij})$ gegeben ist. Die Adjazenzmatrix W enthält bei gewichteten Graphen an der Stelle w_{ij} das Gewicht der Kante (i,j). Existiert diese Kante nicht, so gilt $w_{ij} = \infty$. Des Weiteren gilt $w_{ii} = 0$ für alle $i \in V$.

```
    FLOYD-WARSHALL(W)
1   D^(0) = W
2   for k = 1 to n {
3       Erzeuge (n × n)-Nullmatrix D^(k) = (δ_ij^(k))
4       for i = 1 to n {
5           for j = 1 to n {
6               δ_ij^(k) = min{δ_ij^(k-1), δ_ik^(k-1) + δ_kj^(k-1)}
7           }
8       }
9   }
10  return D^(n)
```

Theorem 4.4.10. *Der Floyd-Warshall-Algorithmus löst das APSP für Graphen ohne Kreise mit negativem Gesamtgewicht, die als Adjazenzmatrix dargestellt sind und n Knoten enthalten, in Zeit* $\Theta(n^3)$.

Beweis. Die Korrektheit des Algorithmus folgt direkt aus der rekursiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben, da der Algorithmus exakt diese rekursive Formel berechnet. Die Laufzeit von $\Theta(n^3)$ ist leicht ersichtlich, da wir drei ineinander geschachtelte **for**-Schleifen haben, die jeweils über n verschiedene Werte zählen, und innerhalb der innersten Schleifen nur Operationen ausgeführt werden, die konstante Zeit benötigen. Das Erstellen der Nullmatrizen benötigt insgesamt ebenfalls Zeit $\Theta(n^3)$. □

Wenn wir zusätzlich zu den Entfernungen auch die kürzesten Wege ermitteln wollen, so können wir dies auch wieder mittels einer rekursiv definierten Vorgängerfunktion π machen. Sei $\pi_{ij}^{(k)}$ der Vorgänger von j auf dem Pfad P_{ij}^k, das heißt auf dem kürzesten Pfad von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k\}$ benutzen darf. Da für $k = 0$ überhaupt keine Zwischenknoten erlaubt sind, gilt für alle Knoten $i, j \in V$

$$\pi_{ij}^{(0)} = \begin{cases} \textbf{null} & \text{falls } i = j \text{ oder } w_{ij} = \infty, \\ \text{i} & \text{falls } i \neq j \text{ und } w_{ij} < \infty. \end{cases}$$

Das heißt, $\pi_{ij}^{(0)}$ ist genau dann gleich i, wenn es eine direkte Kante von i nach j gibt. Für $k > 0$ orientieren wir uns an der rekursiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben. Ist $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$, so ist auch $P_{ij}^k = P_{ij}^{k-1}$ und somit ist der Vorgänger von j auf dem Pfad P_{ij}^k derselbe wie der auf dem Pfad P_{ij}^{k-1}. Ist $\delta_{ij}^{(k)} < \delta_{ij}^{(k-1)}$, so gilt

$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j,$$

und damit ist insbesondere der Vorgänger von j auf dem Pfad P_{ij}^k derselbe wie auf dem Pfad P_{kj}^{k-1}. Zusammenfassend halten wir für $k > 0$ fest

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{falls } \delta_{ij}^{(k-1)} \leq \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{falls } \delta_{ij}^{(k-1)} > \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}. \end{cases}$$

Die $\pi_{ij}^{(k)}$ können zusammen mit den $\delta_{ij}^{(k)}$ im Floyd-Warshall-Algorithmus berechnet werden, ohne dass sich die Laufzeit signifikant ändert.

Negative Kreise Wir wollen nun noch kurz diskutieren, wie man den Floyd-Warshall-Algorithmus benutzen kann, um die Existenz eines Kreises mit negativem Gesamtgewicht zu detektieren. Dazu müssen wir uns nur überlegen, was die Einträge $\delta_{ii}^{(k)}$ auf der Diagonalen der Matrix $D^{(k)}$ bedeuten. Für $k = 0$ werden diese Einträge alle mit 0 initialisiert. Gibt es nun für $k > 0$ einen Kreis mit negativem Gesamtgewicht, der den Knoten i enthält und sonst nur Knoten aus der Menge $\{1, \ldots, k\}$, so gilt $\delta_{ii}^{(k)} < 0$ und somit auch $\delta_{ii}^{(n)} < 0$, da es einen Pfad negativer Länge von i nach i gibt. Also genügt es, am Ende des Algorithmus, die Einträge auf der Hauptdiagonalen zu betrachten. Ist mindestens einer von diesen negativ, so wissen wir, dass ein Kreis mit negativem Gesamtgewicht existiert. In diesem Fall, gibt die vom Algorithmus berechnete Matrix $D^{(n)}$ nicht die korrekten Abstände an.

Vergleich mit Algorithmus von Dijkstra Wir haben weiter oben bereits angesprochen, dass das APSP auch dadurch gelöst werden kann, dass wir einen Algorithmus für das SSSP für jeden möglichen Startknoten einmal aufrufen. Hat der Algorithmus für das SSSP eine Laufzeit von $O(f)$, so ergibt sich zur Lösung des APSP eine Laufzeit von $O(nf)$. Für den Dijkstra-Algorithmus ergibt sich also eine Laufzeit von $O(nm \log n)$.

Für Graphen mit nicht-negativen Kantengewichten stellt sich somit die Frage, ob es effizienter ist, den Floyd-Warshall-Algorithmus mit Laufzeit $O(n^3)$ oder den Dijkstra-Algorithmus mit Lösung $O(nm \log n)$ zur Lösung des APSP zu nutzen. Diese Frage lässt sich nicht allgemein beantworten. Für dichte Graphen mit $m = \Theta(n^2)$ vielen Kanten ist zum Beispiel der Floyd-Warshall-Algorithmus schneller. Für dünne Graphen mit $m = O(n)$ Kanten ist hingegen die n-malige Ausführung des Dijkstra-Algorithmus schneller.

Literatur

Das SSSP und APSP werden in den Kapiteln 24 und 25 von [1] beschrieben. Außerdem sind sie in den Kapiteln 4.3.1 und 4.3.3 von [2] erklärt.

4.5 Flussprobleme

In diesem Abschnitt lernen wir mit Flussproblemen eine weitere wichtige Klasse von algorithmischen Graphproblemen kennen, die in vielen Bereichen Anwendung findet. Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten $s, t \in V$ und einer *Kapazitätsfunktion* $c : V \times V \to \mathbb{N}_0$, die jeder Kante $(u, v) \in E$ eine ganzzahlige nichtnegative Kapazität $c(u, v)$ zuweist. Wir definieren $c(u, v) = 0$ für alle $(u, v) \notin E$. Außerdem nennen wir den Knoten s die *Quelle* und den Knoten t die *Senke*. Wir definieren $n = |V|$ und $m = |E|$. Außerdem nehmen wir an, dass jeder Knoten von der Quelle s zu erreichen ist. Das bedeutet insbesondere, dass $m \geq n - 1$ gilt.

Definition 4.5.1. *Ein* Fluss *in einem Flussnetzwerk ist eine Funktion $f : V \times V \to \mathbb{R}$, die die folgenden beiden Eigenschaften erfüllt.*

1. Flusserhaltung*: Für alle Knoten $u \in V \setminus \{s, t\}$ gilt*

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

2. Kapazitätsbeschränkung*: Für alle Knoten $u, v \in V$ gilt*

$$0 \leq f(u,v) \leq c(u,v).$$

Wir definieren den Wert *eines Flusses $f : V \times V \to \mathbb{R}$ als*

$$|f| := \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

und weisen explizit darauf hin, dass die Notation $|.|$ hier keinen Betrag bezeichnet, sondern den Wert des Flusses gemäß obiger Definition.

Das Flussproblem, das wir uns in diesem Abschnitt anschauen werden, lässt sich nun wie folgt formulieren:

> Gegeben sei ein Flussnetzwerk G. Berechne einen maximalen Fluss auf G, d. h. einen Fluss f mit größtmöglichem Wert $|f|$.

Anschaulich ist die Frage also, wie viele Einheiten Fluss man in dem gegebenen Netzwerk maximal von der Quelle zur Senke transportieren kann.

Als erste einfache Beobachtung halten wir fest, dass der Fluss, der in der Quelle entsteht, komplett in der Senke ankommt und nicht zwischendurch versickert.

Lemma 4.5.2. *Sei f ein Fluss in einem Flussnetzwerk G. Dann gilt*

$$|f| = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v).$$

Beweis. Als erstes beobachten wir, dass die folgenden beiden Summen gleich sind:

$$\sum_{u \in V} \sum_{v \in V} f(v,u) = \sum_{u \in V} \sum_{v \in V} f(u,v).$$

Dies folgt einfach daraus, dass wir sowohl links als auch rechts über jedes Knotenpaar genau einmal summieren. Nun nutzen wir die Flusserhaltung für alle Knoten $u \in V \setminus \{s,t\}$ und erhalten aus obiger Gleichung:

$$\begin{aligned}
&\sum_{u \in \{s,t\}} \sum_{v \in V} f(v,u) = \sum_{u \in \{s,t\}} \sum_{v \in V} f(u,v) \\
\iff &\sum_{v \in V} f(v,s) + \sum_{v \in V} f(v,t) = \sum_{v \in V} f(s,v) + \sum_{v \in V} f(t,v) \\
\iff &\sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) = \sum_{v \in V} f(v,t) - \sum_{v \in V} f(t,v),
\end{aligned}$$

woraus mit der Definition von $|f|$ direkt das Lemma folgt. □

Anwendungsbeispiele

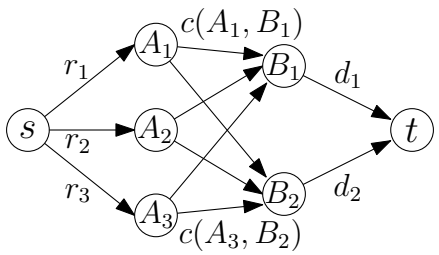
Bevor wir Algorithmen zur Lösung des Flussproblems vorstellen, präsentieren wir ein Anwendungsbeispiel, das wir aus Kapitel 4.5 von [2] übernommen haben. Nehmen wir an, in den Seehäfen $A_1, \ldots, A_p$ liegen Bananen zur Verschiffung bereit. Es gibt Zielhäfen $B_1, \ldots, B_q$, zu denen die Bananen transportiert werden sollen. Für $i = 1, \ldots, p$ bezeichne r_i, wie viele Tonnen Bananen im Hafen A_i bereit stehen, und für $j = 1, \ldots, q$ bezeichne d_j wie viele Tonnen Bananen am Zielfhafen B_j angefordert wurden. Für $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es zwischen den Häfen A_i und B_j eine Schifffahrtslinie, die maximal $c(A_i, B_j)$ Tonnen Bananen transportieren kann. Es kann auch sein, dass es zwischen zwei Häfen A_i und B_j keine Schifffahrtslinie gibt; in diesem Falle definieren wir $c(A_i, B_j)$ als 0. Die folgenden Fragen stellen sich dem Handelsunternehmen:

1. Ist es möglich, alle Anforderungen zu befriedigen?
2. Falls nein, wie viele Tonnen Bananen können maximal zu den Zielhäfen transportiert werden?
3. Wie sollen die Bananen verschifft werden?

Alle diese Fragen können als Flussproblem modelliert werden. Dazu konstruieren wir einen Graphen $G = (V, E)$ mit den Knoten $V = \{s, t, A_1, \ldots, A_p, B_1, \ldots, B_q\}$. Es gibt drei Typen von Kanten:

- Für $i = 1, \ldots, p$ gibt es eine Kante (s, A_i) mit Kapazität r_i.
- Für $j = 1, \ldots, q$ gibt es eine Kante (B_j, t) mit Kapazität q_j.
- Für jedes $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es eine Kante (A_i, B_j) mit Kapazität $c(A_i, B_j)$.

Die folgende Abbildung zeigt ein Beispiel für $p = 3$ und $q = 2$.



Sei $f : E \to \mathbb{R}$ ein maximaler Fluss im Graphen G. Man kann sich anschaulich schnell davon überzeugen, dass man mit Hilfe dieses Flusses alle drei Fragen, die oben gestellt wurden, beantworten kann:

1. Wir können alle Anforderungen befriedigen, wenn der Wert $|f|$ des Flusses gleich $\sum_{j=1}^{q} d_j$ ist.
2. Wir können maximal $|f|$ Tonnen Bananen zu den Zielhäfen transportieren.

3. Der Fluss $f(e)$ auf Kanten $e = (A_i, B_j)$ gibt an, wie viele Tonnen Bananen entlang der Schifffahrtslinie von A_i nach B_j transportiert werden sollen.

Wir verzichten an dieser Stelle auf einen formalen Beweis, dass dies die korrekten Antworten auf obige Fragen sind, möchten aber den Leser dazu anregen, selbst darüber nachzudenken.

Flussprobleme können auch dazu genutzt werden, andere Probleme zu modellieren, die auf den ersten Blick ganz anders geartet sind. Ein Beispiel (siehe Übungen) ist es, zu entscheiden, ob eine Fußballmannschaft bei einer gegebenen Tabellensituation und einem gegebenen restlichen Spielplan gemäß der alten Zwei-Punkte-Regel noch Meister werden kann.

4.5.1 Algorithmus von Ford und Fulkerson

Der folgende Algorithmus von Ford und Fulkerson berechnet für ein gegebenes Flussnetzwerk $G = (V, E)$ mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ einen maximalen Fluss. Wir gehen in diesem Kapitel davon aus, dass das Flussnetzwerk dergestalt ist, dass für kein Paar von Knoten $u, v \in V$ die beiden Kanten (u, v) und (v, u) in E enthalten sind. Dies ist keine wesentliche Einschränkung, da jedes Netzwerk, das diese Eigenschaft verletzt, einfach in ein äquivalentes umgebaut werden kann, das keine entgegengesetzten Kanten enthält. Es ist eine gute Übung für den Leser, sich zu überlegen, wie das funktioniert.

```
    FORD-FULKERSON(G, c, s ∈ V, t ∈ V)
1   Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2   while ∃ flussvergrößernder Weg P {
3       Erhöhe den Fluss f entlang P.
4   }
5   return f;
```

Wir müssen noch beschreiben, was ein *flussvergrößernder Weg* ist und was es bedeutet, den Fluss entlang eines solchen Weges zu erhöhen. Dazu führen wir den Begriff des *Restnetzwerkes* ein.

Definition 4.5.3. *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ und sei f ein Fluss in G. Das dazugehörige* Restnetzwerk *$G_f = (V, E_f)$ ist auf der gleichen Menge von Knoten V definiert wie das Netzwerk G. Wir definieren eine Funktion $\text{rest}_f : V \times V \to \mathbb{R}$ mit*

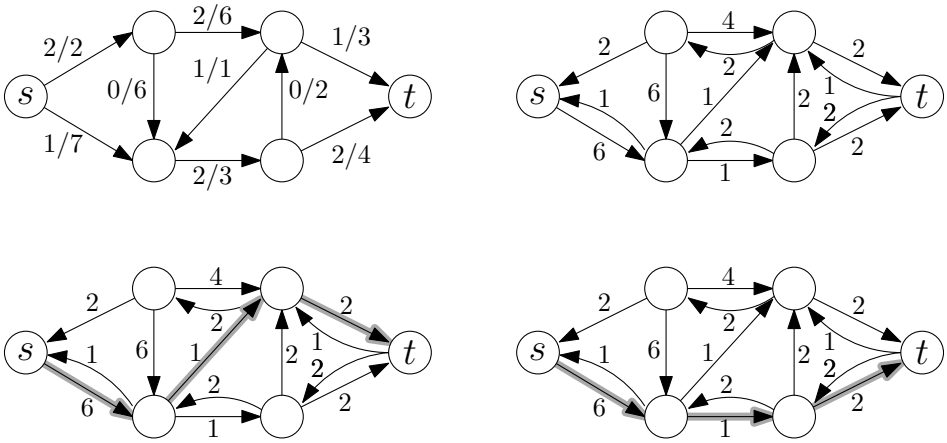
$$\text{rest}_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E, \\ f(v, u) & \text{falls } (v, u) \in E, \\ 0 & \text{sonst.} \end{cases}$$

Die Kantenmenge E_f ist definiert als

$$E_f = \{(u, v) \in V \times V \mid \text{rest}_f(u, v) > 0\}.$$

Ein flussvergrößernder Weg *ist ein einfacher Weg von s nach t im Restnetzwerk G_f.*

Die folgende Abbildung zeigt oben links ein Beispiel für ein Flussnetzwerk mit einem Fluss f. Dabei bedeutet die Beschriftung a/b an einer Kante e, dass $f(e) = a$ und $c(e) = b$ gilt. Rechts oben ist das zugehörige Restnetzwerk dargestellt, und in der unteren Reihe sind zwei flussvergrößernde Wege zu sehen.



Nun müssen wir noch klären, was es bedeutet, den Fluss entlang eines Weges zu erhöhen.

Definition 4.5.4. *Sei G ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$, sei $f : V \times V \to \mathbb{R}$ ein Fluss und sei P ein Weg im Restnetzwerk G_f von s nach t. Wir bezeichnen mit $f \uparrow P : V \times V \to \mathbb{R}$ den Fluss, der entsteht, wenn wir f entlang P erhöhen. Dieser Fluss ist definiert durch*

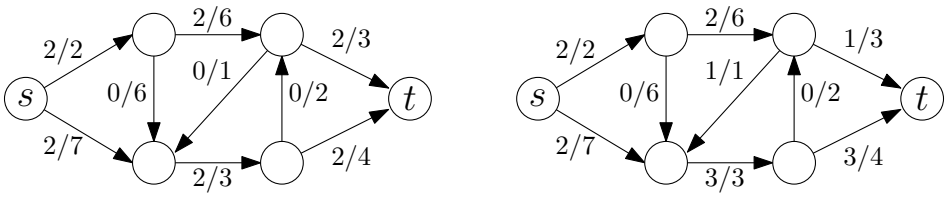
$$(f \uparrow P)(u,v) = \begin{cases} f(u,v) + \delta & \textit{falls } (u,v) \in E \textit{ und } (u,v) \in P, \\ f(u,v) - \delta & \textit{falls } (u,v) \in E \textit{ und } (v,u) \in P, \\ f(u,v) & \textit{sonst,} \end{cases}$$

wobei

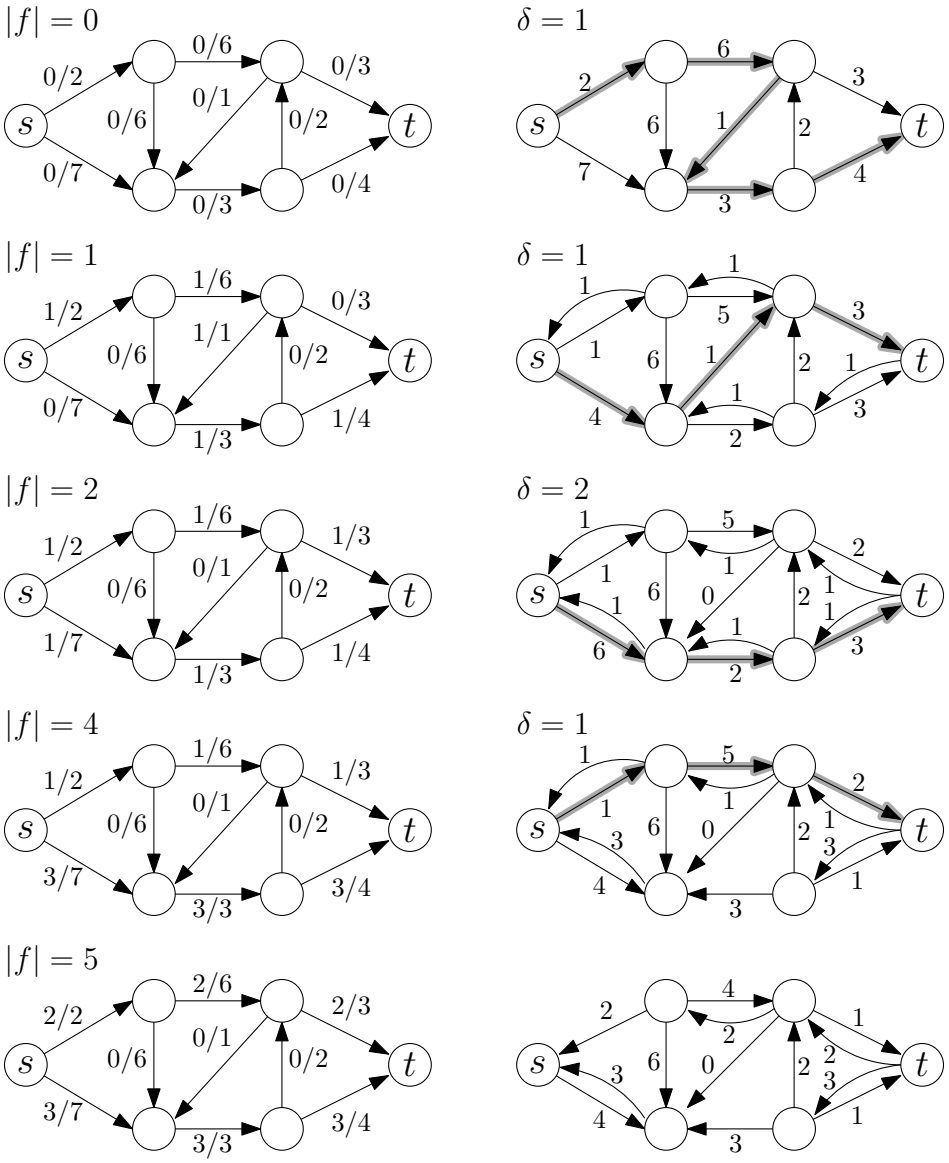
$$\delta = \min_{e \in P}(\mathrm{rest}_f(e)).$$

Aus der Definition von E_f folgt, dass $\delta > 0$ gilt.

In der folgenden Abbildung sind die Flüsse dargestellt, die entstehen, wenn wir den Fluss in obiger Abbildung entlang der beiden oben dargestellten Wege verbessern. In beiden Beispielen ist $\delta = 1$.



Die folgende Abbildung zeigt ein Beispiel für die Ausführung des Algorithmus von Ford und Fulkerson.



Korrektheit

Um die Korrektheit des Algorithmus von Ford und Fulkerson zu zeigen, müssen wir zunächst zeigen, dass $f \uparrow P$ wieder ein Fluss ist.

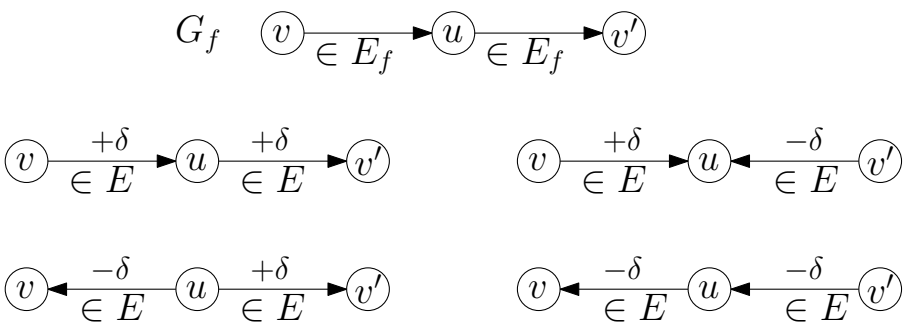
Lemma 4.5.5. *Sei f ein Fluss in einem Netzwerk G und sei P ein Weg von s nach t im Restnetzwerk G_f. Die Funktion $f \uparrow P : V \times V \to \mathbb{R}$ ist wieder ein Fluss. Für diesen Fluss gilt*

$$|f \uparrow P| = |f| + \delta.$$

Beweis. Wir müssen zeigen, dass die beiden Eigenschaften aus Definition 4.5.1 erfüllt sind.

- *Flusserhaltung*: Sei $u \in V \setminus \{s, t\}$. Falls u auf dem Weg P nicht vorkommt, so gilt $f(u, v) = (f \uparrow P)(u, v)$ und $f(v, u) = (f \uparrow P)(v, u)$ für alle Knoten $v \in V$ und somit folgt die Flusserhaltung für u in $f \uparrow P$ aus der Flusserhaltung für u im Fluss f.

Kommt u auf dem Weg P vor, so gibt es genau eine Kante $e \in P$ der Form (v, u) für ein $v \in V$ und es gibt genau eine Kante e' der Form (u, v') für ein $v' \in V$. Die Eindeutigkeit folgt daraus, dass P per Definition einfach ist und somit den Knoten u nur einmal besucht. Die Kanten e und e' sind die einzigen zu u inzidenten Kanten, auf denen sich der Fluss ändert. Wir brauchen also nur diese beiden Kanten näher zu betrachten. Es gilt entweder $(v, u) \in E$ oder $(u, v) \in E$. Ebenso gilt entweder $(v', u) \in E$ oder $(u, v') \in E$. Wir schauen uns alle vier Fälle einzeln an und überprüfen, dass die Flusserhaltung erhalten bleibt. In der folgenden Abbildung ist oben der relevante Ausschnitt des Restnetzwerkes dargestellt und darunter die vier Fälle. In jedem der Fälle ist leicht zu sehen, dass die Flusserhaltung an u erhalten bleibt.



- *Kapazitätsbeschränkung*: Seien $u, v \in V$ beliebig so gewählt, dass $e = (u, v) \in E_f$ eine Kante auf dem Weg P in G_f ist. Wir zeigen, dass $0 \leq (f \uparrow P)(e) \leq c(e)$ gilt. Dafür unterscheiden wir zwei Fälle.
  - Ist $e = (u, v) \in E$, so erhöhen wir den Fluss auf e um δ. Auf Grund der Definition von δ und wegen $f(e) \geq 0$ und $\delta \geq 0$ gilt
  $$0 \leq f(e) + \delta \leq f(e) + \text{rest}_f(e) = f(e) + (c(e) - f(e)) = c(e).$$
  - Ist $e' = (v, u) \in E$, so verringern wir den Fluss auf e' um δ. Auf Grund der Definition von δ und wegen $f(e') \leq c(e')$ und $\delta \geq 0$ gilt
  $$c(e') \geq f(e') - \delta \geq f(e') - \text{rest}_f(e) \geq f(e') - f(e') = 0.$$ □

Aus Lemma 4.5.5 folgt direkt, dass die Funktion f, die der Algorithmus von Ford und Fulkerson verwaltet, nach jeder Iteration ein Fluss ist. Wir zeigen nun, dass der Algorithmus, wenn er terminiert, einen maximalen Fluss berechnet hat. Dazu führen wir zunächst die folgende Definition ein.

Definition 4.5.6. *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$. Sei $s \in V$ die Quelle und $t \in V$ die Senke. Ein* Schnitt *von G ist eine Partitionierung der Knotenmenge V in zwei Teile $S \subseteq V$ und $T \subseteq V$ mit $s \in S$, $t \in T$ und $T = V \setminus S$. Wir nennen*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

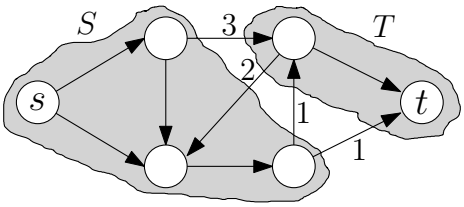
die Kapazität *des Schnittes (S, T). Ein Schnitt (S, T) heißt* minimal*, wenn es keinen Schnitt (S', T') mit $c(S', T') < c(S, T)$ gibt.*

Für einen Fluss f und einen Schnitt (S, T) sei

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u)$$

der Fluss über den Schnitt.

Die folgende Abbildung zeigt einen Schnitt mit Kapazität 5.



Anschaulich ist leicht einzusehen, dass die Kapazität $c(S,T)$ eine obere Schranke dafür ist, wie viel Fluss von der Quelle zur Senke geschickt werden kann, und dass für einen Fluss f der Fluss über den Schnitt (S,T) genau $|f|$ sein muss. Formal zeigen wir das in folgendem Lemma.

Lemma 4.5.7. *Sei (S,T) ein Schnitt eines Flussnetzwerkes G und sei f ein Fluss in G. Dann gilt*

$$|f| = f(S,T) \leq c(S,T).$$

Beweis. Ähnlich zu dem Beweis von Lemma 4.5.2 summieren wir zunächst über alle Kanten, die innerhalb der Menge S verlaufen. Wir erhalten die folgende Gleichung:

$$\begin{aligned} &\sum_{v\in S} f(s,v) + \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(u,v) = \sum_{v\in S} f(v,s) + \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(v,u) \\ \iff &\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(v,u) - \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(u,v). \end{aligned} \tag{4.2}$$

Für alle Knoten $u \in S \setminus \{s\}$ gilt die Flusserhaltung. Dafür müssen wir aber auch die Kanten betrachten, die von u in die Menge T führen, und die Kanten, die von der Menge T zu u führen. Wir erhalten

$$\sum_{v\in S} f(u,v) + \sum_{v\in T} f(u,v) = \sum_{v\in S} f(v,u) + \sum_{v\in T} f(v,u).$$

Damit können wir (4.2) schreiben als

$$\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u). \tag{4.3}$$

Für die Quelle s gilt entsprechend

$$\begin{aligned} &\sum_{v\in S} f(s,v) + \sum_{v\in T} f(s,v) = |f| + \sum_{v\in S} f(v,s) + \sum_{v\in T} f(v,s) \\ \iff &\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = |f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v). \end{aligned} \tag{4.4}$$

Wir setzen (4.4) in (4.3) ein und erhalten

$$\begin{aligned} &|f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v) = \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u) \\ \iff &|f| = \sum_{u\in S} \sum_{v\in T} f(u,v) - \sum_{u\in S} \sum_{v\in T} f(v,u) = f(S,T). \end{aligned}$$

Die Ungleichung folgt einfach aus obiger Gleichung, da $f(u,v) \leq c(u,v)$ und $f(v,u) \geq 0$ für alle $u, v \in V$:

$$f(S,T) \leq \sum_{u\in S} \sum_{v\in T} f(u,v) \leq \sum_{u\in S} \sum_{v\in T} c(u,v) \leq c(S,T). \qquad \square$$

Mit Hilfe von Lemma 4.5.7 zeigen wir das folgende Theorem, aus dem direkt folgt, dass der Algorithmus von Ford und Fulkerson einen maximalen Fluss gefunden hat, wenn er terminiert.

Theorem 4.5.8. *Sei f ein Fluss in einem Netzwerk G. Dann sind die folgenden drei Aussagen äquivalent.*

a) f ist ein maximaler Fluss.

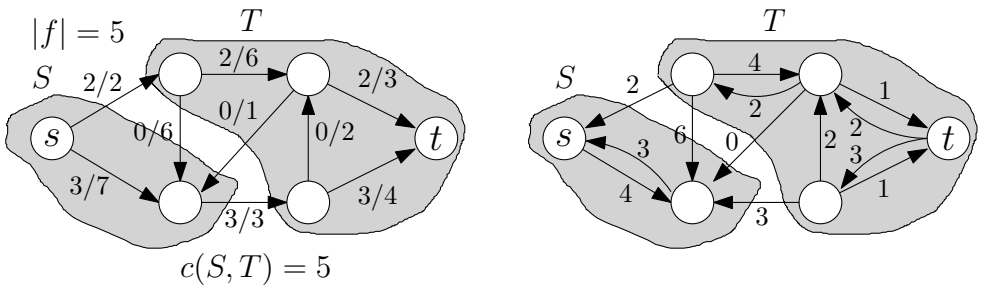
b) Das Restnetzwerk G_f enthält keinen flussvergrößernden Weg.

c) Es gibt einen Schnitt (S,T) mit $|f| = c(S,T)$.

Dieses Theorem besagt insbesondere, dass es einen Schnitt (S,T) geben muss, dessen Kapazität so groß ist wie der Wert $|f|$ des maximalen Flusses f. Zusammen mit Lemma 4.5.7, das besagt, dass es keinen Schnitt (S',T') mit kleinerer Kapazität als $|f|$ geben kann, folgt die wichtige Aussage, dass die Kapazität des minimalen Schnittes gleich dem Wert des maximalen Flusses ist. Die Korrektheit des Algorithmus von Ford und Fulkerson bei Terminierung folgt einfach daraus, dass der Algorithmus erst dann terminiert, wenn es keinen flussvergrößernden Weg mehr gibt. Nach dem Theorem ist das aber genau dann der Fall, wenn der Fluss maximal ist.

Beweis von Theorem 4.5.8. Wir werden zeigen, dass b) aus a) folgt, dass c) aus b) folgt und dass a) aus c) folgt. Diese drei Implikationen bilden einen Ringschluss und damit ist gezeigt, dass die drei Aussagen äquivalent sind.

- **a) ⇒ b)** Wir führen einen Widerspruchsbeweis und nehmen an, dass es einen flussvergrößernden Weg P in G_f gibt. Dann können wir den Fluss f entlang P erhöhen und erhalten den Fluss $f \uparrow P$, welcher nach Lemma 4.5.5 einen um $\delta > 0$ höheren Wert als f hat. Damit kann f kein maximaler Fluss sein.
- **c) ⇒ a)** Sei f ein Fluss mit $|f| = c(S,T)$ für einen Schnitt (S,T) und sei f' ein maximaler Fluss. Dann gilt $|f| \leq |f'|$. Wenn wir Lemma 4.5.7 für den Fluss f' und den Schnitt (S,T) anwenden, dann erhalten wir $|f'| \leq c(S,T)$. Zusammen ergibt das
$$c(S,T) = |f| \leq |f'| \leq c(S,T),$$
woraus $c(S,T) = |f| = |f'|$ folgt. Damit ist auch f ein maximaler Fluss.
- **b) ⇒ c)** Laut Voraussetzung gibt es keinen flussvergrößernden Weg, das heißt, es gibt keinen Weg von s nach t im Restnetzwerk G_f. Wir teilen die Knoten V des Restnetzwerkes in zwei Klassen S und T ein:
$$S = \{v \in V \mid \text{es gibt Weg in } G_f \text{ von } s \text{ nach } t\} \quad \text{und} \quad T = V \setminus S.$$
Da der Knoten t in G_f nicht von s aus erreichbar ist, gilt $s \in S$ und $t \in T$. Damit ist (S,T) ein Schnitt des Graphen. Die Situation ist in folgender Abbildung dargestellt.



Wir argumentieren nun, dass $|f| = c(S,T)$ für diesen Schnitt (S,T) gilt.

- Sei $(u,v) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(u,v) = c(u,v)$.
- Sei $(v,u) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(v,u) = 0$.

Somit gilt mit Lemma 4.5.7

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) = f(S,T) = |f| \leq c(S,T).$$

Damit folgt $|f| = f(S,T) = c(S,T)$. □

Laufzeit

Wir haben den Korrektheitsbeweis noch nicht abgeschlossen. Wir haben lediglich gezeigt, dass der Algorithmus *partiell korrekt* ist. Das bedeutet, wir haben gezeigt, dass der Algorithmus das richtige Ergebnis liefert, falls er terminiert. Wir haben aber noch nicht gezeigt, dass er wirklich auf jeder Eingabe terminiert. Das erledigen wir nun implizit bei der Laufzeitabschätzung.

Theorem 4.5.9. *Für ganzzahlige Kapazitäten $c : V \times V \to \mathbb{N}_0$ ist die Anzahl an Iterationen der* **while***-Schleife im Algorithmus von Ford und Fulkerson durch $C = \sum_{e \in E} c(e)$ nach oben beschränkt. Die Laufzeit des Algorithmus ist $O(mC)$.*

Beweis. Man kann sehr einfach (z. B. mit Hilfe einer Invariante) zeigen, dass der aktuelle Fluss f, den der Algorithmus verwaltet stets ganzzahlig ist, dass also stets $f : V \times V \to \mathbb{N}_0$ gilt. Dies folgt einfach daraus, dass für einen ganzzahligen Fluss f und einen flussvergrößernden Weg P stets $\delta \in \mathbb{N}$ gilt. Demzufolge steigt der Wert des Flusses mit jeder Iteration der **while**-Schleife um mindestens eins an. Da der Wert eines jeden Flusses durch $\sum_{e \in E} c(e)$ beschränkt ist, garantiert dies die Terminierung nach höchstens C vielen Schritten.

Für einen gegebenen Fluss f kann das Restnetzwerk G_f in Zeit $O(m)$ berechnet werden. Ein Weg P von s nach t in G_f kann mittels Tiefensuche in Zeit $O(m)$ gefunden werden. Ist ein solcher Weg P gefunden, so kann in Zeit $O(m)$ der neue Fluss $f \uparrow P$ berechnet werden. Insgesamt dauert also jede Iteration Zeit $O(m)$, woraus das Theorem folgt. □

Aus diesem Theorem und seinem Beweis folgt direkt, dass es bei ganzzahligen Kapazitäten auch stets einen ganzzahligen maximalen Fluss gibt. Dies ist eine sehr wichtige Eigenschaft für viele Anwendungen und deswegen halten wir sie explizit als Korollar fest.

Korollar 4.5.10. *Sind alle Kapazitäten ganzzahlig, so gibt es stets einen ganzzahligen maximalen Fluss.*

4.5.2 Algorithmus von Edmonds und Karp

Die Laufzeit $O(mC)$, die wir in Theorem 4.5.9 bewiesen haben, ist nicht besonders gut, weil sie mit den Kapazitäten der Kanten wächst. Dies steht im Gegensatz zu den Laufzeiten von zum Beispiel den Algorithmen von Kruskal und Dijkstra. Für diese Algorithmen haben wir Laufzeitschranken bewiesen, die nur von der Größe des Graphen abhängen, nicht jedoch von den Gewichten oder Längen der Kanten. Eine Laufzeit wie $O(mC)$, die von den Kapazitäten abhängt, heißt *pseudopolynomielle Laufzeit*. Wir werden diesen Begriff im nächsten Semester weiter vertiefen. Hier sei nur angemerkt, dass man, wenn möglich, Laufzeiten anstrebt, die nur von der Größe des Graphen, nicht jedoch von den Kapazitäten abhängen, da diese sehr groß werden können. Wir werden nun den *Algorithmus von Edmonds und Karp* kennenlernen. Dabei handelt es sich um eine leichte Modifikation des Algorithmus von Ford und Fulkerson, deren Laufzeit nur von der Größe des Graphen abhängt.

Wir haben in der Beschreibung des Algorithmus von Ford und Fulkerson offen gelassen, wie die flussvergrößernden Wege gewählt werden, wenn es mehrere zur Auswahl gibt. Unsere Analyse von Korrektheit und Laufzeit war unabhängig von dieser Wahl. Wir zeigen nun, dass für eine geeignete Wahl die Laufzeit reduziert werden kann.

```
    EDMONDS-KARP(G, c, s ∈ V, t ∈ V)
1   Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2   while ∃ flussvergrößernder Weg P {
3       Wähle einen flussvergrößernden Weg P mit so wenig Kanten wie möglich.
4       Erhöhe den Fluss f entlang P.
5   }
6   return f;
```

Der einzige Unterschied zum Algorithmus von Ford und Fulkerson ist also, dass immer ein kürzester flussvergrößernder Weg gewählt wird, wobei die Länge eines Weges als die Anzahl seiner Kanten definiert ist. Wir werden in diesem Zusammenhang auch von der *Distanz von s zu $v \in V$ im Restnetzwerk G_f* sprechen. Damit meinen wir, wie viele Kanten der kürzeste Weg (d. h. der Weg mit den wenigsten Kanten) von s nach v in G_f hat.

Theorem 4.5.11. *Die Laufzeit des Algorithmus von Edmonds und Karp ist $O(m^2n) = O(n^5)$.*

Beweis. Eine Iteration der **while**-Schleife kann immer noch in Zeit $O(m)$ durchgeführt werden. Einen kürzesten flussvergrößernden Weg findet man nämlich zum Beispiel, indem man von s startend eine Breitensuche im Restnetzwerk durchführt. Diese benötigt Zeit $O(m)$. Zum Beweis des Theorems genügt es also zu zeigen, dass der Algorithmus nach $O(mn)$ Iterationen terminiert. Dazu zeigen wir zunächst die folgende strukturelle Eigenschaft über die Distanzen im Restnetzwerk.

Lemma 4.5.12. *Sei $x \in V$ beliebig. Die Distanz von s zu x in G_f wird im Laufe des Algorithmus nicht kleiner.*

Beweis. Wir betrachten eine Iteration der **while**-Schleife, in der der Fluss f entlang des Weges P zu dem Fluss $f \uparrow P$ erhöht wird. Die Kantenmenge $E_{f\uparrow P}$ unterscheidet sich von der Kantenmenge E_f wie folgt:

- Für jede Kante $(u, v) \in P \subseteq E_f$ verringert sich $\text{rest}_f(u, v)$ um δ, das heißt $\text{rest}_{f\uparrow P}(u, v) = \text{rest}_f(u, v) - \delta$. Eine Kante mit $\text{rest}_f(u, v) = \delta$ heißt *Flaschenhalskante* und sie ist in $E_{f\uparrow P}$ nicht mehr enthalten.

- Für jede Kante $(u, v) \in P \subseteq E_f$ erhöht sich die Restkapazität $\text{rest}_f(v, u)$ der entgegengesetzten Kante um δ, das heißt $\text{rest}_{f\uparrow P}(v, u) = \text{rest}_f(v, u) + \delta$. War $\text{rest}_f(v, u) = 0$, so war $(v, u) \notin E_f$, nun gilt aber $(v, u) \in E_{f\uparrow P}$.

Wir spalten den Übergang von E_f zu $E_{f\uparrow P}$ in mehrere Schritte auf. Zunächst fügen wir alle neuen Kanten nach und nach ein, die gemäß dem zweiten Aufzählungspunkt entstehen. Anschließend entfernen wir die Kanten nach und nach gemäß dem ersten Aufzählungspunkt, um die Kantenmenge $E_{f\uparrow P}$ zu erhalten. Nach jedem Einfügen und Löschen einer Kante untersuchen wir, wie sich die Distanz von s zu x geändert hat.

Als erstes betrachten wir den Fall, dass wir eine Kante (v, u) einfügen. Gemäß dem zweiten Aufzählungspunkt bedeutet das, dass die Kante (u, v) auf dem Weg P liegen muss. In dem Algorithmus wurde der Weg P so gewählt, dass er ein kürzester Weg von s nach t ist. Wegen der optimalen Substruktur, die wir uns für kürzeste Wege überlegt haben, bedeutet das insbesondere, dass die Teilwege von P, die von s nach u und v führen, kürzeste Wege von s nach u bzw. v sind. Die Länge des Teilweges nach u bezeichnen wir mit ℓ. Dann ist die Länge des Teilweges nach v genau $\ell + 1$. Somit verbindet die neue Kante einen Knoten mit Distanz $\ell + 1$ von s mit einem Knoten mit Distanz ℓ von s. Dadurch kann sich jedoch die Distanz von s zu keinem Knoten des Graphen verringern. Um die Distanz zu verringern, müsste die neue Kante einen Knoten mit Distanz k von s für ein $k \geq 0$ mit einem Knoten mit Distanz $k + i$ von s für ein $i \geq 2$ verbinden. Damit folgt, dass selbst nach dem Einfügen aller neuen Kanten der Knoten x immer noch die gleiche Distanz von s hat.

Es ist leicht einzusehen, dass das Löschen von Kanten keine Distanzen verringern kann. Deshalb hat der Knoten x nach dem Löschen mindestens die gleiche Distanz von s wie in G_f. □

Mit Hilfe von Lemma 4.5.12 können wir nun das Theorem beweisen. Bei jeder Iteration der **while**-Schleife gibt es mindestens eine Flaschenhalskante (u, v). Diese wird aus dem Restnetzwerk entfernt. Sei ℓ die Distanz von s zu u, bevor die Kante entfernt wird. Die Distanz zu v ist dann $\ell + 1$. Es ist möglich, dass diese Kante zu einem späteren Zeitpunkt wieder in das Restnetzwerk eingefügt wird, aber nur dann, wenn die Kante (v, u) in einer späteren Iteration auf dem gewählten flussvergrößernden P' liegt. Da sich die Distanz von s zu v nicht verringern kann, muss zu diesem Zeitpunkt die Distanz von s nach v immer noch mindestens $\ell + 1$ sein. Da P' ein kürzester Weg ist, können wir wegen der optimalen Substruktur wieder folgern, dass die Distanz von s zu u zu diesem Zeitpunkt mindestens $\ell + 2$ sein muss.

Zusammenfassend können wir also festhalten, dass das Entfernen und Wiedereinfügen einer Kante (u, v) im Restnetzwerk die Distanz von s zu u um mindestens 2 erhöht. Da die maximale Distanz $n - 1$ ist, kann eine Kante demnach nicht öfter als $n/2$ mal entfernt werden. In jeder Iteration wird mindestens eine Kante entfernt und es gibt $2m$ potentielle Kanten im Restnetzwerk. Damit ist die Anzahl an Iterationen der **while**-Schleife nach oben beschränkt durch $\frac{n}{2} \cdot 2m = nm$. □

Literatur

Flussprobleme werden in Kapitel 26 von [1] beschrieben. Außerdem sind sie in Kapitel 4.5 von [2] erklärt.

Literaturverzeichnis

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. MIT Press, 3. Auflage, 2009.

[2] Norbert Blum. Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2004.