Effiziente Algorithmen 1 - Zusammenfassung

Patrick Dammann

21.05.2017

Zweck	Name	Laufzeit	Kommentar	Link
Durchsuchen	**Breitensuche**	$\mathcal{O}(\text{Adj.})\star$	Laufzeit je nach Adjazenzstruktur	2.2.1 auf Seite 6
	Tiefensuche	$\mathcal{O}(\text{Adj.})\star$		2.2.2 auf Seite 6
Topologische Sortierung	**Topsort**	$\mathcal{O}(\text{Adj.})\star$	findet auch Kreise	2.3 auf Seite 7
Komponenten finden	**StrongComponents**	$\mathcal{O}(\text{Adj.})\star(?)$		2.4 auf Seite 7
Minimum Spanning Tree	**Boruvka**	¯_(ツ)_/¯	paralleler Ansatz	3.1.2 auf Seite 8
	Kruskal	$\mathcal{O}(E \log V)$	Kanten sortieren	3.1.3 auf Seite 9
	Prim	–	baut MST von Startknoten	3.1.4 auf Seite 9
		$\mathcal{O}(VE)$	Implementierung	3.1.4 auf Seite 9
		$\mathcal{O}(V^2)$ $\mathcal{O}(E \log V)$ $\mathcal{O}(E + V \log V)$	Bessere Impl., sucht sinnvoller	3.1.4 auf Seite 9
	Round Robin	$\mathcal{O}(E \log \log V)$	baut MST von überall	3.1.5 auf Seite 10
1-Baum	**OneTree**	-> MST	Untere Schranke TSP	3.1.6 auf Seite 10
Maximales Branching	**Branching**	$\mathcal{O}(A \log V)$ oder $\mathcal{O}(V^2)$	Schrumpfen:	3.2.1 auf Seite 11 3.2.1 auf Seite 11
Kürzeste Wege	**Dijkstra**	$\mathcal{O}(V^2)$ $\mathcal{O}(A \log V)$ $\mathcal{O}(A + V \log V)$	Keine neg. Kanten	4.2 auf Seite 12
	BellmanFord	$\mathcal{O}(VA)$		4.3 auf Seite 12
	DAGShortestPath	$\mathcal{O}(V + A)$	Nur in DAGs	4.4 auf Seite 13
Alle kürzesten Wege	FloydWarshall	$\mathcal{O}(V^3)$	Wege über erste k Knoten	4.5.1 auf Seite 14
	Johnson	$\mathcal{O}(V \times \text{Dijkstra})$	1×BF, $\|V\|$ ×Dijkstra	4.5.1 auf Seite 14
Kreise mit bestem KZV	<kein Name>	$\mathcal{O}(VA \log(CVT))$	μ schätzen	
Minimales Matching	**Ungarische Methode**	$\mathcal{O}(n^3)$		5.2 auf Seite 15
	AKP-Algorithmus	$\mathcal{O}(n^3)$		5.7 auf Seite 17
Maximaler (s,t)-Fluss	**FordFulkerson**	$\mathcal{O}(A \lvert f^* \rvert)$	a.W. im reduzierten Netzwerk	6.1 auf Seite 20
	EdmondsKarp	$\mathcal{O}(VA^2)$	FF mit wenig Kanten auf a.W.	6.3 auf Seite 20
	ScalingMaxFlow	$\mathcal{O}(A^2 \log C)$ $\mathcal{O}(VA \log C)$	nur Kanten mit $c_f > K$	6.4 auf Seite 21
	Preflow-Push (FIFO-Variante)	$\mathcal{O}(V^2 A)$ $\mathcal{O}(V^3)$	Push, Lift (Examine)	6.17 auf Seite 21 6.5 auf Seite 22
Minimaler Schnitt	**NagamochiIbaraki**	$\mathcal{O}(V^3)$ $\mathcal{O}(V^2 + VE \log V)$	legale Ordnung	7.1 auf Seite 23
	Karger	¯_(ツ)_/¯	MC-Algo, $p = \frac{2}{n(n-1)}$	7.2 auf Seite 24
Netzwerkfluss	**SteppingStone**	¯_(ツ)_/¯		
	Neg.-Kreise-Verfahren	$\mathcal{O}(V^2 AUW)$	zul. Fluss wird kostenminimiert	8.3 auf Seite 26
	Kürz.-Wege-Verfahren	$\mathcal{O}(BV^2)$	kostenmin. Fluss wird zulässig	8.4 auf Seite 27
		$\mathcal{O}(BA \log V)$	auch ohne dass **alle** KW existieren	8.4 auf Seite 27
	CapacityScaling	$\mathcal{O}(AV^2 \log U)$ $\mathcal{O}(A^2 \log V \log U)$	analog zu ScalingMaxFlow	8.6 auf Seite 27

$\star\mathcal{O}(V^2)$ für Matrix, $\mathcal{O}(V + E)$ für Liste

1 Probleme und Algorithmen

Lineares kombinatorisches Optimierungsproblem

Gegeben sind eine endliche Menge E, ein System von Teilmengen $\mathcal{I} \subseteq 2^E$ (zulässige Lösungen) und eine Funktion $c : E \to \mathbb{R}$. Es ist eine Menge $I^* \in \mathcal{I}$ zu bestimmen, so dass $c(I^*) = \sum_{e \in I^*} c(e)$ minimal bzw. maximal ist.

Euklidisches Traveling-Salesman-Problem

Gegeben sind n Punkte in der Euklidischen Ebene. Zu bestimmen ist eine geschlossene Tour, die jeden Punkt genau einmal besucht und möglichst kurz ist.
E = Menge der Kanten
$\mathcal{I}$ = Alle Mengen von Kanten, die eine Tour bilden

Euklidisches Matching-Problem

Gegeben sind n Punkte in der Euklidischen Ebene (n gerade). Zu bestimmen sind $\frac{n}{2}$ Linien, so dass jeder Punkt Endpunkt genau einer Linie ist und die Summe der Linienlängen so klein wie möglich ist.
E = Menge der Kanten
$\mathcal{I}$ = Alle Mengen von Kanten mit der Eigenschaft, dass jeder Knoten zu genau einer der Kanten gehört.

Einheitskosten-Modell Es werden nur die Schritte des Algorithmus gezählt, die Zahlengrößen bleiben unberücksichtigt.

Bit-Modell Die Laufzeit für eine arithmetische Operation ist M, wobei M die größte Kodierungslänge einer an dieser Operation beteiligten Zahl ist.

Definition 1.1. Die Laufzeitfunktion $f_A : \mathbb{N} \to \mathbb{N}$ ist in $\mathcal{O}(g)$ für eine Funktion $g : \mathbb{N} \to \mathbb{N}$ falls es eine Konstante $c > 0$ und $n_o \in \mathbb{N}$ gibt, so dass $f_A \leq c \cdot g(n)$ für alle $n \geq n_o$.

Definition 1.2. Ein Algorithmus heißt **effizient** bzw. **polynomialer Algorithmus**, wenn seine Laufzeit in $\mathcal{O}\left(n^k\right)$ liegt. Ein Problem, das mit einem polynomialen Algorithmus gelöst werden kann, heißt **polynomiales Problem**.

Definition. Ein **Graph** G ist ein Tupel $G = (V, E)$[1] bestehend aus einer nicht-leeren Knotenmenge V und einer Kantenmenge E.

- Ein Graph heißt **endlich**, wenn V und E endlich sind.
- Wenn $e = \{u, v\} \in E$ und $u, v \in V$, dann sind u und v **Nachbarn** bzw. **adjazen**t, sind **Endknoten** von e und werden von e **verbunden**.
- Eine Kante $e = \{u, u\} \in E$ heißt **Schleife**.
- Kanten mit $E \ni e = \{u, v\} = f \in E$ heißen **parallel** oder **mehrfach**.
- Ein Graph ohne Mehrfachkanten heißt **einfach**.
- Für $W \subseteq V$ bekommt die Menge aller Knoten in $V \setminus W$ mit Nachbarn in W die Bezeichnung $\Gamma(W)$.
- Kurzform von $\Gamma(\{v\})$ ist $\Gamma(v)$.
- Die Menge $\delta(W)$ aller Kanten mit je einem Endknoten in W und $V \setminus W$ heißt **Schnitt**.
- Kurzform von $\delta(\{v\})$ ist $\delta(v)$.
- Der **Grad** eines Knoten v ist die Anzahl seiner Nachbarn, bzw. $|\delta(v)|$.
- Ein **(s,t)-Schnitt** ist ein Schnitt $\delta(W)$ mit $s \in W$ und $t \in V \setminus W$ und gleichzeitig ein (t,s)-Schnitt.
- Mit $W \subseteq V$ ist $E(W)$ die Menge aller Kanten mit beiden Endknoten in W.
- Mit $F \subseteq E$ ist $V(F)$ die Menge aller Knoten, die Endknoten von mind. einer Kante in F sind.
- Sind $G = (V, E)$ und $H = (W, F)$ Graphen und $W \subseteq V$ und $F \subseteq E$, so heißt H **Untergraph** von G.

[1] In der Vorlesung werden primär endliche, einfache, schleifenfreie Graphen behandelt, die der Einfachheit halber eine Notation ohne Inzidenzfunktion nutzen können. In diesem Skript wird (sofern nicht anders angegeben) von solchen Graphen ausgegangen.

- Mit $W \subseteq V$ ist $G - W$ der Graph G ohne die Knoten in W und ohne alle Kanten an W.
- $G[W] = G - (V \setminus W)$ ist der **von W induzierte Untergraph**.
- Mit $F \subseteq E$ ist $G - F = (V, E \setminus F)$.
- Kurzform von $G - \{x\}$ ist $G - x$ für $x \in E$ oder $x \in V$.
- Ein einfacher Graph heißt **vollständig**, wenn jede mögliche Kante zwischen seinen Knoten existiert.
- Der vollstängige Graph mit n Knoten wird mit $K_n = (V_n, E_n)$ bezeichnet.
- Das **Komplement** des Graphen $G = (V, E)$ ist $\bar{G} = (V, E_n \setminus E)$.
- Ein Graph heißt **bipartit**, wenn er sich in zwei disjunkte Teilmengen V_1, V_2 mit $V_1 \cup V_2 = V$ teilen lässt, ohne dass es Kanten $\{u, v\}$ mit $u, v \in V_1 \vee u, v \in V_2$ gibt.

Definition. Ein **Digraph** G ist ein Tupel $D = (V, A)$ bestehend aus einer nicht-leeren Knotenmenge V und einer Kantenmenge A.

- Wenn $a = (u, v) \in A$ und $u, v \in V$, dann ist u **Anfangsknoten** und v **Endknoten** von a. Hier heißt u **Vorgänger** von v und v **Nachfolger** von u.
- Die Kanten (u, v) und (v, u) heißen **antiparallel**.
- Mit $W \subseteq V$ ist $A(W)$ die Menge aller Kanten mit Anfangs- und Endknoten in W.
- Mit $B \subseteq A$ ist $V(A)$ die Menge aller Knoten, die Anfangs- oder Endknoten von mind. einer Kante in B sind.
- $G = (V, E)$ ist der unterliegende Graph von $D = (V, A)$, wenn E genau die Kanten $\{u, v\}$ enthält, für die (u, v) oder (v, u) in A liegen.
- Ein einfacher Digraph heißt **vollständig**, wenn jede mögliche Kante (in beide Richtungen) zwischen seinen Knoten existiert.
- Der vollstängige Digraph mit n Knoten wird mit $D_n = (V_n, A_n)$ bezeichnet.
- Für $W \subseteq V, W \neq V \neq \emptyset$ enthält $\delta^+(W) = \{(i, j) \in A \mid i \in W, j \notin W\}$ alle Kanten, die W verlassen, $\delta^-(W) = \delta^+(V \setminus W)$ alle Kanten, die in W hinein führen und $\delta(W) = \delta^+(W) \cup \delta^-(W)$ beide. Diese Mengen heißen **Schnitt**. Es gelten die Kurzformen für einzelne Knoten.
- Für $s \in W$ und $t \notin W$ heißt $\delta^+(W)$ auch (s, t)**-Schnitt**.
- Die Kardinalitäten der Schnitte heißen **Außengrad** ($|\delta^+(v)|$), **Innengrad** ($|\delta^-(v)|$) und **Grad** ($|\delta(v)|$).
- $\delta^+(W)$ heißt **gerichteter Schnitt**, wenn $\delta^-(W) = \emptyset$.

Definition. Eine endliche Folge $W = (v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k)$ heißt **Kette** oder $[v_0, v_k]$-Kette der Länge k, wenn jede Kante e_i die Knoten v_{i-1} und v_i in einem (Di-)Graphen indiziert und **gerichtete Kette** oder (v_0, v_k)-Kette, wenn alle Kanten in der Form $e_i = (v_{i-1}, v_i)$ sind. v_0 und v_k heißen **Anfangs- und Endknoten**.

- Gibt es in einem (Di-)Graphen keine parallelen Kanten, ist eine (gerichtete) Kette durch ihre Knoten eindeutig identifiziert.
- Ein **(gerichteter) Weg** oder **(gerichteter) Pfad** ist eine (gerichtete) Kette, in der alle Knoten verschieden sind.
- Die Notation $u \underset{D}{\to} v$ bedeutet, dass es einen (u, v)-Weg in D gibt.
- Man spricht auch von (u, v)-Wegen bzw. (u, v)-Pfaden.
- Die Knoten s und t eines Graphen heißen **zusammenhängend**, wenn ein (s, t)-Weg existiert.
- Ein **zusammenhängender Graph** enthält nur Knoten, die paarweise zusammenhängend sind.
- Ein Digraph heißt **stark zusammenhängend**, wenn es zu jedem Knotenpaar s, t einen (s, t)-Weg und einen (t, s)-Weg gibt.

- **Kompontenten** eines Graphen sind die (bezüglich Kanteninklusion) maximalen zusammenhängenden Untergraphen.
- **Starke Kompontenten** eines Diraphen sind die (bezüglich Kanteninklusion) maximalen stark zusammenhängenden Unterdigraphen.
- Ein Graph heißt k**-fach zusammenhängend**, wenn jedes Paar Knoten s, t durch mindestens k (s,t)-Wege verbunden ist, die keine inneren Knoten gemeinsam haben.
- Ein Digraph heißt k**-fach stark zusammenhängend**, wenn jedes Paar Knoten s, t durch mindestens k (s,t)-Wege und (t,s)-Wege verbunden ist, die keine inneren Knoten gemeinsam haben.
- Eine **geschlossene Kette** hat mehr als 0 Kanten und den gleichen Anfangs- und Endknoten.
- Ein **Kreis** ist eine geschlossene Kette mit paarweise verschiedenen inneren Knoten. Seine **Länge** ist die Anzahl seiner Kanten.
- Ein **Eulerpfad** ist eine Kette, die jede Kante eines (Di-)Graphen einmal enthält.
- Eine **Eulertour** ist ein geschlossener Eulerpfad.
- Ein **Eulergraph** ist ein Graph, der eine Eulertour enthält.
- Ein **Hamiltonkreis** (oder **Hamiltontour**) ist ein Kreis der Länge $|V|$.
- Ein **hamiltonischer** Graph enthält einen Hamiltonkreis.
- Ein **Hamiltonweg** ist ein (gerichteter) Weg der Länge $|V| - 1$.
- Ein **Wald** ist eine Kantenmenge in einem Graphen, die keinen Kreis enthält.
- Ein **Baum** ist ein zusammenhängender Wald.
- Ein **aufspannender** Baum enthält alle Knoten des Graphen.
- Ein **azyklischer** Digraph enthält keine gerichteten Kreise.
- Ein **Branching** in einem Digraphen ist eine azyklische Kantenmenge, sodass jeder Knoten maximal eine Eingangskante besitzt.
- Eine **Arboreszenz** ist ein zusammenhängendes Branching.
- Eine **aufspannende Arboreszenz** enthält alle Knoten ihres Digraphen.
- **Gewichte** (auch "Kosten", "Distanzen", "Kapazitäten", usw.) werden durch Funktionen der Form $c : E \to \mathbb{R}$ bzw. $c : A \to \mathbb{R}$ mit Kanten assoziiert.

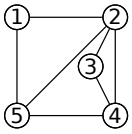
2 Grundlegende Graphenalgorithmen

2.1 Repräsentationen von Graphen

Adjazenzliste Jeder Knoten hat eine Liste seiner Nachbarn gespeichert. Speichersparend für dünne Graphen. Hinzufügen und Entfernen von Knoten und Kanten sehr einfach. Existenz von Kanten prüfen teuer (einen Knoten durchlaufen). Speicheraufwand: $\mathcal{O}(V + E)$

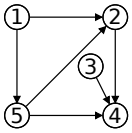
Adjazenzmatrix $|V| \times |V|$-Matrix, die an der Stelle (u, v) eine 1 hat, wenn die Kante (u, v) bzw. $\{u, v\}$ existiert. Existenz von Kanten prüfen in $\mathcal{O}(1)$. Nachbarn durchlaufen unanhängig von ihrer Anzahl in $\mathcal{O}(V)$. Speicheraufwand: $\mathcal{O}\left(V^2\right)$

Inzidenzmatrix $|V| \times |E|$-Matrix, die an der Stelle (v, e) eine 1 hat, wenn der Knoten v Endknoten der Kante e ist. Erlaubt Untersuchung des Graphen mit diversen, algebraischen Methoden. *[Kein Beispiel]*



$$\begin{bmatrix} 1 & \to 2 & \to 5 & & \\ 2 & \to 1 & \to 3 & \to 4 & \to 5 \\ 3 & \to 2 & \to 4 & & \\ 4 & \to 2 & \to 3 & \to 5 & \\ 5 & \to 1 & \to 2 & \to 4 & \end{bmatrix}$$

$$\left[\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}\right]$$



$$\begin{bmatrix} 1 & \to 2 & \to 5 \\ 2 & \to 4 & \\ 3 & \to 4 & \\ 4 & & \\ 5 & \to 2 & \to 4 \end{bmatrix}$$

$$\left[\begin{array}{c|ccccc} \nearrow & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 1 & 0 \end{array}\right]$$

2.2 Durchsuchen von Graphen

Das "färben" von Knoten ist eine Kurzschreibweise für folgende Sachverhalte:

weiß Der Knoten ist noch nicht erreicht. (Im Normalfall Grundzustand)

grau Der Knoten wurde erreicht, seine Nachbarn jedoch noch nicht abgearbeitet.

schwarz Der Knoten wurde komplett bearbeitet.

2.2.1 Breitensuche (BFS)

Start bei Knoten s. $\pi[u] = v$ heißt v ist direkter Vorgänger von u. Q ist eine Queue.

BFS(G, s):

(1) Färbe s grau, setze $d[s] = 0$, $\pi[s] = \perp$ und initialisiere Q mit s.

(2) Färbe alle Knoten $v \in V \setminus \{s\}$ weiß und setze $d[v] = +\infty$ und $\pi[v] = \perp$.

(3) Falls Q leer, *Stop*("BFS beendet"), sonst sei u erster Knoten in Q.

(4) Für alle v aus Adjazenzliste von u:

(4.1) Falls v weiß ist, färbe v grau, setze $d[v] = d[u] + 1$, $\pi[v] = u$ und füge v ans Ende von Q ein.

(5) Entferne u aus Q, färbe u schwarz und gehe zu (3).

Laufzeit: linear in Bezug auf Adjazenzstruktur. Das heißt $\mathcal{O}(V + E)$ bei Adjazenzlisten und $\mathcal{O}(V^2)$ bei Adjazenzmatrizen.

Definition. Für $v \in V$ sei $\delta(s, u)$ die Zahl der Kanten des kürzesten (s, u)-Weges, bzw. ∞ wenn kein solcher existiert.

Lemma 2.1. *Sei $G = (V, E)$ ein Graph und $s \in V$.*

- *Für jede Kante $uv \in E$ gilt: $\delta(s, v) \leq \delta(s, u) + 1$.*
- *Nach Terminierung von BFS(G, s) gilt: $\forall v \in V : d[v] \geq \delta(s, v)$*
- *Enthält Q während BFS(G, s) $v_1, v_2, \ldots, v_r$ gilt: $d[v_r] \leq d[v_1] + 1$ und $d[v_i] \leq d[v_{i+1}], 1 \leq i < r$*

Satz 2.2. *Sei $G = (V, E)$, $s \in V$ und BFS(G, s) ausgeführt. Dann ist jeder Knoten, der von s aus erreichbar ist, schwarz gefärbt und es gilt $d[v] = \delta(s, v)$.*

2.2.2 Tiefensuche (DFS)

Kein spezieller Startknoten. Die Zeit der Grau-Färbung wird in $d[v]$ gespeichtert, die Zeit der Schwarz-Färbung (Terminierungszeit) in $t[v]$.

DFS(D):

(1) Färbe alle Knoten $u \in V$ weiß und setze $\pi[u] = \perp$.

(2) Setze globale Zeit $t = 0$.

(3) Für jeden Knoten $u \in V$:

(3.1) Falls u weiß gefärbt ist, dann DFSVisit(u).

DFSVisit(u):

(1) Färbe u grau, setze $t = t + 1$ und $d[u] = t$.

(2) Für alle v aus Adjazenzliste von u:

(2.1) Falls v weiß ist, dann setze $\pi[v] = u$ und vollziehe DFSVisit(v).

(3) Färbe u schwarz, setze $t = t + 1$ und $f[u] = t$.

Laufzeit: linear in Bezug auf Adjazenzstruktur. Das heißt $\mathcal{O}(V + A)$ bei Adjazenzlisten und $\mathcal{O}(V^2)$ bei Adjazenzmatrizen.

Definition. Der **DFS-Wald** des (Di-)Graphen $G = (V, E)$ ist ein Digraph der Form $(V, \{\{\pi[v], v\} | v \in V, \pi[v] \neq \perp\})$. Er symbolisiert also den Weg durch den Graphen während einer DFS.
Alle Kanten $\{u, v\}$ aus E, die nicht zum Wald gehören sind:

Vorwärtskanten Es gibt einen (u, v)-Weg im DFS-Wald

Rückwärtskanten Es gibt einen (v, u)-Weg im DFS-Wald

Kreuzungskanten Es gibt keinen der Wege im DFS-Wald

Für ungerichtete Graphen gibt es nur Rückwärtskanten.

Satz 2.3. *Ein Knoten v ist Nachfolger eines Knotens u im DFS-Wald genau dann, wenn gilt: Zu dem Zeitpunkt, zu dem u grau gefärbt wird, ist v von u aus auf einem Weg erreichbar, der nur aus weißen Knoten besteht.*

2.3 Topologisches Sortieren

Die topologische Sortierung eines Digraphen ist eine Knotenreihenfolge, in der alle Kanten nur zu später auftretenden Knoten führen.

Topsort(D):

(1) Initialisiere leere Liste $L = \emptyset$.

(2) Führe DFS(D) aus, mit Zusatz: Wenn ein Knoten v schwarz gefärbt wird, füge ihn am Anfang der Liste L ein.

(3) L liefert topologische Sortierung.

Laufzeit: siehe DFS(D)

Lemma 2.4. *Ein Digraph D ist genau dann kreisfrei, wenn DFS(D) keine Ruckwärtskanten liefert.*

Satz 2.5. *Ein azyklischer Digraph D wird durch Topsort(D) topologisch sortiert.*

2.4 Starke Zusammenhangskomponenten

StrongComponents(D):

(1) Führe DFS(D) aus, merke Terminierungszeiten $f[u]$ für alle $u \in V$.

(2) Generiere $D^\top$.

(3) Führe DFS$\left(D^\top\right)$ aus, wobei im Schritt (3) die Knoten nach absteigenden Werten von $f[u]$ sortiert werden.

(4) Die Knoten jedes Baumes im DFS-Wald von $D^\top$ bestimmen eine starke Zusammenhangskomponente.

Lemma 2.6. *Jeder Weg, der zwei Knoten aus derselben Komponente verbindet, enthält nur Knoten aus dieser Komponente.*

Lemma 2.7. *Alle Knoten einer Komponente sind im gleichen DFS-Baum enthalten.*

Definition. Für einen Knoten u bezeichne $\phi(u)$ den Knoten v mit der größten Terminierungszeit bei DFS(D), der von u aus in D erreichbar ist, das heißt:
$\phi(u) = \text{argmax}_v \left\{ f[v] \,|\, u \underset{D}{\rightarrow} v \right\}$

Satz 2.8. *Für $D = (V, A)$ sei DFS(D) ausgeführt. Dann ist für jeden Knoten $\phi(u)$ Vorgänger von u im DFS-Wald.*

Satz 2.9. *Nach DFS(D) liegen zwei Knoten u und v genau dann in der gleichen Komponente, wenn $\phi(u) = \phi(v)$.*

Satz 2.10. *Der Algorithmus StrongComponents identifiziert die starken Zusammenhangskomponenten eines Digraphen.*

3 Optimale Bäume und Branchings

3.1 Minimale aufspannende Bäume

Die Probleme, einen minimalen, aufspannenden Baum oder einen maximalen, aufspannenden Wald zu finden lassen sich einfach ineinander transformieren. Darum beschränken wir uns auf ersteres.

Minimaler-aufspannender-Baum-Problem
Gegeben ist ein Graph $G = (V, E)$ mit Kantengewichten c_e für $e \in E$. Zu bestimmen ist ein aufspannender Baum für G, dessen Gesamtgewicht möglichst klein ist.
(Wir nehmen an, dass G zusammenhängend ist, da das Problem sonst einzeln pro Komponente lösbar wäre.)

3.1.1 Ein allgemeiner MST-Algorithmus

Dieser Algorithmus färbt Kanten nach festen Regeln blau und rot. Es ist stets die Invariante *„Es gibt einen minimalen, aufspannenden Baum, der alle blauen und keine rote Kante enthält.“* erfüllt.

Regel B Wähle einen Schnitt, der keine blaue Kante enthält. Färbe eine seiner kürzesten Kanten blau.

Regel R Wähle einen Kreis, der keine rote Kante enthält. Färbe eine seiner längsten Kanten rot.

MinimumSpanningTree(G, c) :

(1) Zu Beginn seien alle Kanten ungefärbt.

(2) Wende Regeln B und R an, bis alle Kanten gefärbt sind.

(3) Die blauen Kanten bilden einen minimalen aufspannenden Baum.

Satz 3.1. *Der Algorithmus färbt alle Kanten eines zusammenhängenden Graphen und erhält dabei die Invarianz-Bedingung, d.h. er berechnet einen minimalen aufspannenden Baum.*

3.1.2 Der Algorithmus von Boruvka

Boruvka(G, c):

(1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.

(2) Solange mehr als ein blauer Baum vorhanden ist, wähle gleichzeitig zu jedem blauen Baum die minimale inzidente Kante und färbe alle ausgewählten Kanten blau.

(3) Die blauen Kanten bilden einen minimalen Baum.

Der Algorithmus kann blaue Kreise erzeugen, wenn es im Graphen Kanten gleichen Gewichts gibt.

3.1.3 Der Algorithmus von Kruskal

Kruskal(G, c):

(1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.

(2) Sortiere die Kanten von G nach nichtabsteigenden Gewichten, so dass $c_{e_1} \leq c_{e_2} \leq \ldots \leq c_{e_m}$.

(3) Für $i = 1, 2, \ldots, m$:

(3.1) Sind die Endknoten von e_i im gleichen blauen Baum, färbe e_i rot, andernfalls blau.

(4) Die blauen Kanten bilden einen minimalen Baum.

Laufzeit: $\mathcal{O}(E \log V)$, wenn die Bäume per fast-union-find verwaltet werden. Ein Heap ermöglicht Optimierung bezöglich der Sortierung, da abgebrochen werden kann, wenn es $n-1$ blaue Kanten gibt.

3.1.4 Der Algorithmus von Prim

Der Algorithmus verwendet nur Regel B und benötigt einen Startknoten s.

Prim(G, c, s):

(1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.

(2) Solange noch blaue Bäume aus einem Knoten existieren, färbe minimale Kante im Schnitt induziert durch den Baum, der s enthält, blau.

(3) Die blauen Kanten bilden einen minimalen Baum.

Es folgt eine einfach Implementierung. T ist der angehende Baum:

Prim1(G, c, s):

(1) Setze $V_T = \{s\}$, $T = \emptyset$ und $i = 0$.

(2) Falls $i = n - 1$, *Stop*(„T ist MST“).

(3) Bestimme $u \in V_T$ und $v \in V \setminus V_T$ mit $c_{uv} = \min\{c_{xy} | x \in V_T, y \in V \setminus V_T\}$.

(4) Setze $V_T = V_T \cup \{v\}$ und $T = T \cup \{uv\}$ und $i = i + 1$. Gehe zu (2).

Laufzeit: $\mathcal{O}(VE)$

Es gibt eine bessere Implementierung.$t[v]$ speichert den nächsten Baumknoten, $d[v]$ die zugehörige Distanz.

Prim2(G, c, s):

(1) Setze $V_T = \{s\}$, $T = \emptyset$ und $i = 0$. Setze $d[v] = c_{sv}$ und $t[v] = s$, falls $sv \in E$, setze $d[v] = \infty$ und $t[v] = \perp$, falls $sv \notin E$.

(2) Falls $i = n - 1$, *Stop*(„T ist MST“).

(3) Bestimme $v \in V \setminus V_T$ mit $d[v] = \min\{d[u] | u \in V \setminus V_T\}$.

(4) Setze $V_T = V_T \cup \{v\}$ und $T = T \cup \{t[v]\, v\}$.

(5) Für alle w adjazent zu v:

(5.1) Falls $w \in V \setminus V_T$ und $c_{vw} < d[w]$, dann setze $d[w] = c_{vw}$ und $t[w] = v$.

(6) Setze $i = i + 1$ und gehe zu (2).

Laufzeit (naiv):	$\mathcal{O}(V^2)$
Laufzeit (Heap):	$\mathcal{O}(E \log V)$ (besser für dünnbesetzte Graphen)
Laufzeit (Fib-Heap):	$\mathcal{O}(E + V \log V)$

3.1.5 Der Round-Robin-Algorithmus

RoundRobin(G, c):

(1) Initialisiere n blaue Bäume bestehend aus je einem Knoten.

(2) Solange weniger als $n-1$ blaue Kanten vorhanden sind, wähle einen blauen Baum, bestimme die kürzeste Kante im durch diesen Baum induzierten Schnitt und färbe sie blau.

(3) Die blauen Kanten bilden einen minimalen Baum.

Eine $\mathcal{O}(E \log\log V)$-Implementation ist möglich, wenn immer der kleinste Baum gewählt wird.

3.1.6 Eine Anwendung aufspannender Bäume

Sogenannte 1-Bäume sind eine untere Schranke für eine optimale TSM-Tour.

OneTree(G, c):

(1) Bestimme für die Knoten $\{2, 3, \ldots, n\}$einen MST T. Sei c_T die Länge des MST.

(2) Seien e_1 und e_2 die zwei kürzesten Kanten an Knoten 1.

(3) $T \cup \{e_1, e_2\}$ist optimaler 1-Baum mit Wert $c_T + c_{e_1} + c_{e_2}$.

3.2 Maximale Branchings

Neue Notationen: $s : A \to V$, $t : A \to V$ und $c : A \to \mathbb{R}$ sind Start- und Endknoten sowie Gewichte von Kanten.

Maximales-Branching-Problem
Gegeben ist ein Digraph $D = (V, A)$ mit Kantengewichten c_e, für $e \in A$. Zu bestimmen ist ein Branching für D, dessen Gesamtgewicht möglichst groß ist.

3.2.1 Der Branching-Algorithmus von Edmonds

Definition 3.2. Sei $D = (V, A)$ ein Digraph mit Kantengewichten c_e, für $e \in A$.

a) Eine Kante $e \in A$ heißt **kritisch**, falls $c(e) > 0$ und falls $c(e') \leq c(e)$, für alle $e' \in A$ mit $t(e') = t(e)$.

b) Ein Subgraph $H \subseteq A$ heißt **kritisch**, falls er nur aus kritischen Kanten besteht, jeder Knoten Endknoten von höchstens einer dieser Kanten ist, und falls er inklusionsmaximal bezüglich dieser Eigenschaft ist.

Lemma 3.3. *Ein azyklischer kritischer Graph H ist ein maximaler Branching.*

Lemma 3.4. *Sei H ein kritischer Graph. Dann ist jeder Knoten von H in maximal einem Kreis enthalten.*

Lemma 3.5. *Seien B ein Branching und u, v, w drei Knoten. Falls $u \underset{B}{\to} v$ und $w \underset{B}{\to} v$, dann gilt entweder $u \underset{B}{\to} w$ oder $w \underset{B}{\to} u$.*

Definition. Sei B ein Branching. Eine Kante $e \notin B$ heißt **zulässig** (relativ zu B), falls die Menge $B' = B \cup \{e\} \setminus \{f | f \in B \wedge t(f) = t(e)\}$ ebenfalls ein Branching ist.

Lemma 3.6. *Sei B ein Branching und $e \in A \setminus B$. Dann ist e genau dann zulässig relativ zu B, wenn kein Weg von $t(e)$ nach $s(e)$ in B existiert.*

Lemma 3.7. *Sei B ein Branching und C ein Kreis mit der Eigenschaft, dass keine Kante aus $C \setminus B$ zulässig relativ zu B ist. Dann gilt $|C \setminus B| = 1$.*

Satz 3.8. *Sei H ein kritischer Graph. Dann existiert ein Branching B mit maximalem Gewicht, so dass für jeden Kreis $C \subseteq H$ gilt $|C \setminus B| = 1$.*

Im Folgenden bezeichne $V_i = V(C_i)$ und a_i die kürzeste Kante in C_i.

Korollar 3.9. *Sei $D = (V, A)$ ein Digraph mit Kantengewichten und H ein kritischer Graph mit Kreisen $C_i, i = 1, 2, \ldots, k$. Es existiert ein gewichtsmaximales Branching B mit den Eigenschaften*

a) $|C_i \setminus B| = 1$, *für* $i = 1, 2, \ldots, k$

b) *Falls für jede Kante $e \in B \setminus C_i$ gilt, dass $t(e) \notin V_i$,dann folgt $C_i \setminus B = \{a_i\}$.*

Schrumpfen von kritischen Graphen:
Sei $\widetilde{V} = V \setminus \bigcup_{i=1}^{k} V_i$. Für $v \in V_i$ sei $\widetilde{e}(v)$ die Kante aus C_i mit $t(\widetilde{e}(v)) = v$, d.h. die Kreiskante mit Endknoten v.

$$\begin{aligned}
\overline{A} =& \{e \in A | \text{für kein } i \text{ gilt } s(e) \in V_i \text{ und } t(e) \in V_i\} \\
=& A \setminus \bigcup_{i=1}^{k} A(V(C_i)), \\
\overline{V} =& \widetilde{V} \cup \{w_1, w_2, \ldots, w_k\}, \text{ wobei die } w_i \text{ neue Symbole sind,} \\
\overline{s}(e) =& \begin{cases} s(e), & \text{falls } s(e) \in \widetilde{V}, \\ w_i & \text{falls } s(e) \in V_i, \end{cases} \\
\overline{t}(e) =& \begin{cases} t(e), & \text{falls } t(e) \in \widetilde{V}, \\ w_i & \text{falls } t(e) \in V_i, \end{cases} \\
\overline{c}(e) =& \begin{cases} c(e), & \text{falls } t(e) \in \widetilde{V}, \\ c(e) - c(\widetilde{e}(t(e))) + c(a_i) & \text{falls } t(e) \in V_i. \end{cases}
\end{aligned}$$

Es entsteht ein neues Problem $\overline{P}$ im Digraphen $\overline{D} = (\overline{V}, \overline{A})$ mit den neuen Inzidenzfunktionen $\overline{s}$, $\overline{t}$ und Gewichtsfunktion $\overline{c}$.

Satz 3.10. *Es gibt eine bijektive Abbildung zwischen $\mathcal{B}$ (der Menge der Branchings im Originalproblem) und der Menge der Branchings im Problem $\overline{P}$. Es korrespondiert das Branching $B \in \mathcal{B}$ mit dem Branching $\overline{B} = B \cap \overline{A}$ in $\overline{P}$ und es gilt:*

$$c(B) - \overline{c}(\overline{B}) = \sum_{i=1}^{k} c(C_i) - \sum_{i=1}^{k} c(a_i).$$

Branching(D, c):

(1) Bestimme einen kritischen Graphen H für D.

(2) Ist H azyklisch, dann *Stop*(„H ist optimales Branching für D“).

(3) Schrumpfe die Kreise von H, um $\overline{D}$ und $\overline{c}$ zu erhalten.

(4) Berechne durch den rekursiven Aufruf *Branching*$(\overline{D}, \overline{c})$ ein optimales Branching $\overline{B}$ für $\overline{D}$.

(5) Expandiere das Branching $\overline{B}$ zu einem optimalen Branching B für D.

Laufzeit: $\mathcal{O}(A \log V)$ oder $\mathcal{O}(V^2)$.

3.2.2 Arboreszenzen und das asymmetrische TSP

→analog zum symmetrischen Fall

4 Kürzeste Wege

Kürstester-(s,t)-Weg-Problem
Gegeben sind ein Digraph $D = (V, A)$ mit Kantengewichten c_e für $e \in A$, und zwei Knoten $s, t \in V$. Zu bestimmen ist ein (s,t)-Weg in D, dessen Gesamtlänge möglichst klein ist.

4.1 Eigenschaften kürzester Wege

Definition. Im folgenden ist $s \in V$ der Startknoten, dessen kürzeste Wege berechnet werden sollen. $\delta(u,v)$ ist die Länge des kürzesten (u,v)-Weges, bzw. $+\infty$ (oder $-\infty$) wenn keiner existiert. $\pi[v]$ ist der Vorgängerknoten von v auf dem kürstesten (s,v)-Weg.

Lemma 4.1. *Sei $p = (v_1v_2, v_2v_3, \ldots, v_{k-1}v_k,)$ ein kürzester Weg von v_1 nach v_k. Dann ist für $1 \leq i \leq j \leq k$ der Teilweg $(v_iv_{i+1}, \ldots, v_{j-1}v_j,)$ ein kürzester Weg von v_i nach v_j.*

Lemma 4.2. *Für jede Kante (u,v) gilt $\delta(s,v) \leq \delta(s,u) + c_{uv}$.*

Die **Standardinitialisierung** für d ist $d[s] = 0$ und $d[v] = \infty, v \in V \setminus \{s\}$ und $\pi[v] = \perp, v \in V$.

Correct(u,v):
Falls $d[v] > d[u] + c_{uv}$, dann setze $d[v] = d[u] + c_{uv}$ und $\pi[v] = u$.

Lemma 4.3. *Nach Initialisierung gilt $d[v] \geq \delta(s,v)$, für alle $v \in V$. Dies gilt auch nach Ausführung einer beliebigen Folge von Correct()-Aufrufen. Falls einmal $d[v] = \delta(s,v)$ gilt, so kann $d[v]$nicht mehr verändert werden.*

Lemma 4.4. *D enthalte keine von s aus erreichbaren negativen Kreise und die Standardinitialisierung sei ausgeführt. Für jede Folge von Korrekturoperationen bildet der durch π bestimmte Digraph D_π eine Arboreszenz mit Wurzel s.*

4.2 Der Algorithmus von Dijkstra

Der Algorithmus berechnet alle kürzesten Wege von s, wenn $c_{uv} \geq 0$, für alle $(u,v) \in A$.

Dijkstra(D,c,s):

(1) Setze $d[v] = \infty$, für alle $v \in V \setminus \{s\}$, $d[s] = 0$ und $\pi[v] = \perp$ für alle $v \in V$.

(2) $S = \emptyset, Q = V$.

(3) Solange $Q \neq \emptyset$:

(3.1) Bestimme u mit $d[u] = \min_v \{d[v] \,|v \in Q\}$.

(3.2) $S = S \cup \{u\}, Q = Q \setminus \{u\}$

(3.3) Für jede Kante $(u,v) \in A$ mit $v \in Q$ führe *Correct*(u,v) aus.

Laufzeit (naiv):	$\mathcal{O}(V^2)$
Laufzeit (Heap):	$\mathcal{O}(A \log V)$ (besser für dünnbesetzte Graphen)
Laufzeit (Fib-Heap):	$\mathcal{O}(A + V \log V)$

Satz 4.5. *Nach Ausführung des Algorithmus von Dijkstra gilt $d[v] = \delta(s,v)$ für alle $v \in V$.*

4.3 Der Algorithmus von Bellman und Ford

BellmanFord(D,c,s):

(1) Initialisiere $d[v] = \infty$, für alle $v \in V \setminus s$, $d[s] = 0, \pi[v] = \perp$ für alle $v \in V$.

(2) Führe die folgende Schleife $(|V|-1)$-mal aus:

(2.1) Für alle $(u,v) \in A$ führe *Correct*(u,v) aus.

(3) Für alle $(u,v) \in A$:

(3.1) Falls $d[v] > d[u] + c_{uv}$ dann *Stop*(„Graph enthält negativen Kreis“).

(4) *Stop*(„Der Kürzeste-Wege-Baum wurde berechnet“).

Laufzeit: $\mathcal{O}(VA)$

Lemma 4.6. *D enthalte keinen von s aus erreichbaren negativen Kreis. Dann gilt bei Terminierung des Bellman-Ford-Algorithmus $d[v] = \delta(s,v)$ für alle Knoten in v, die von s aus erreichbar sind.*

Korollar 4.7. *Es gibt einen Weg von s nach v genau dann, wenn der Bellman-Ford-Algorithmus mit $d[v] < \infty$ terminiert.*

Satz 4.8. *Der Bellman-Ford-Algorithmus arbeitet korrekt, das heißt entweder er stellt fest, dass D einen negativen Kreis enthält, oder er berechnet die Arboreszenz der kürzesten Wege von s zu allen erreichbaren Knoten.*

Yen-Variante Ersetze (2.1) durch:

(2.1) Für $i = 1, \ldots, n$: *Correct*(v_i, v_k) für alle $(v_i, v_k) \in A$ mit $i < k$.

(2.2) Für $i = n, n-1, \ldots, 1$: *Correct*(v_i, v_j) für alle $(v_i, v_j) \in A$ mit $i > j$.

Variante von d'Esopo und Pape Bevorzugt Knoten, die sich verbessert haben. Exponentielle Worst-Case-Laufzeit, aber oft effizienter für dünne Graphen.
Modifikation der Correct-Operation:

Correct(u, v):
Falls $d[v] > d[u] + c_{uv}$, dann:

(1) Setze $d[v] = d[u] + c_{uv}$ und $\pi[v] = u$.

(2) Falls v noch nicht in Q war, setze v an das Ende von Q.

(3) Falls v schon in Q war, aber gegenwärtig nicht Q ist, dann setze v an den Anfang von Q.

Terminiert, wenn Q leer.

Verfahren von Nicholson Sucht kürzesten (s, t)-Weg gleichzeitig von s und t.

A^*-Verfahren Verwendet Schätzwerte $s(v, t)$ für die Entfernung zum Zielknoten, priorisiert Knoten mit niedrigem $d[v] + s(v, t)$. Terminiert, wenn $d[t]$ und $s(s, t)$ nag genug beieinander.

4.4 Kürzeste Wege in azyklischen Digraphen

DAGShortestPath(D, c, s):

(1) Sei $t[1], \ldots, t[n]$ eine topologische Sortierung der Knoten von D.

(2) Setze $d[v] = \infty$, für alle $v \in V \setminus \{s\}$, $\pi[v] = \perp$, für alle $v \in V$, und $d[s] = 0$.

(3) Für $i = 1, 2, \ldots, n$:

(3.1) Setze $u = t[i]$.

(3.2) Für alle $v \in V$ mit $(u, v) \in A$ führe *Correct*(u, v) *aus.*

Laufzeit: $\mathcal{O}(V + A)$

4.4.1 Berechnung optimaler Lösungen des Knapsack-Problems

4.4.2 Eine Anwendung in der Tourenplanung

4.4.3 Längste Wege

Dies ist nur in azyklischen Digraphen möglich, oder wenn alle Kantengewichte negativ oder 0 sind.
Entweder man komplementiert alle Gewichte oder ändert die Initialisierung von d in $-\infty$ und dreht den Vergleich in der *Correct*-Operation um.

4.5 Kürzeste Wege zwischen allen Knotenpaaren

Eine Möglichkeit wäre das $|V|$-malige Anwenden von Dijkstra oder Bellman-Ford, was in der $|V|$-fachen Komplexität resultiert.

4.5.1 Der Algorithmus von Floyd und Warshall

D habe ganzzahlige Gewichte und keine negativen Kreise.

AllPairsShortestPaths(D, c):

(1) Setze $d[i, j] = +\infty$, für alle $i, j \in V$, und $d[i, i] = 0$, für alle $i \in V$.

(2) Für alle $(i, j) \in A$, setze $d[i, j] = c_{ij}$.

(3) Solange es Knoten $i, j, k \in V$ gibt, mit $d[i, j] > d[i, k] + d[k, j]$ setze $d[i, j] = d[i, k] + d[k, j]$.

Laufzeit: $\mathcal{O}\left(|V|^2 (2|V| C)\right)$ mit $C = \max\{|c_{ij}| \,|\, (i, j) \in A\}$ -> nicht poly!
Eine Verbesserung ergibt sich wie folgt:

Wenn alle kürzesten Wege bekannt sind, die nur die ersten $k-1$ Knoten als Zwischenknoten verwenden, lässt sich ein kürzester Weg, der den k-ten Knoten (und möglicherweise seine Vorgänger) als Zwischenknoten nutzt durch $d(i,k)+d(k,j)$ bestimmen.
Sei $D^{(k)}$ eine Matrix mit $d_{ij}^{(k)}$ als kürzester Weg zwischen i und j, der nur die ersten k Knoten als Zwischenknoten nutzt:

FloydWarshall(C):

(1) Initialisiere $D^{(0)} = C$, mit $c_{ij} = +\infty$ wenn keine Kante (i,j) existiert.

(2) Für $k = 1, 2, \ldots, |V|$:

(2.1) Für $i = 1, 2, \ldots, |V|$:

(2.1.1) Für $j = 1, 2, \ldots, |V|$:

$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$$

(3) $D^{(|V|)}$ enthält die Längen der kürzesten Wege.

Laufzeit: $\mathcal{O}\left(V^3\right)$

Johnson(D, c):

(1) Bilde $D' = (V', A')$ mit $V' = V \cup \{s\}$, $A' = A \cup \{(s,v) \,|v \in V\}$ und setze $c_{sv} = 0$, für alle $v \in V$.

(2) Führe *Bellman-Ford*(D', c, s) aus.

(3) Enthält D' negative Kreise, *Stop*(„Algorithmus nicht anwendbar“)

(4) Setze für jeden Knoten $v \in V : h[v] = \delta(s,v)$.

(5) Setze für jede Kante $(u,v) \in A : \overline{c}_{uv} = c_{uv} + h[u] - h[v]$.

(6) Für jeden Knoten $u \in V$:

(6.1) Führe Dijkstra$(D, \overline{c}, u)$ aus. Sei $\overline{\delta}(u,v)$ die Kürzeste-Wege-Distanz von u nach $v \in V \setminus \{u\}$ (bzgl. $\overline{c}$).

(6.2) Für jedes $v \in V \setminus \{u\}$ ist die Länge des kürzesten (u,v)-Weges $\overline{\delta}(u,v) + h[v] - h[u]$.

Laufzeit: $\mathcal{O}(VA \log V)$ oder $\mathcal{O}\left(VA + V^2 \log V\right)$ je nach Dijkstra-Implementierung

4.6 Varianten kürzester Wege

4.6.1 Bottleneck-Probleme

In einem Graphen wird ein (s,t)-Weg gesucht, dessen kürzeste Kante größtmögliche Länge hat.

Initialisierung $d[v] = -\infty$, für alle $v \in V \setminus \{s\}$, $d[s] = +\infty, \pi[v] = \perp$, für alle $v \in V$.

Korrektur Falls $d[v] < \min\{d[u], c_{uv}\}$, dann setze $d[v] = \min\{d[u], c_{uv}\}$ und $\pi[v] = u$.

4.6.2 Netzwerke mit Gewinnen und Verlusten

Das **Arbitrage-Problem** funktioniert wie ein Kürzeste-Wege-Problem, nur dass Kanten auf einem Weg multipliziert werden. Entweder Initialisierung und Correct werden angepasst, oder alle Kantengewichte logarithmiert.

4.7 Kreise mit bestem Kosten-Zeit-Verhältnis

Gesucht ist $\mu^* = \min_{B \text{ Kreis in } D} \mu(B)$ mit $\mu(B) = \frac{\sum_{(i,j)\in B} c_{ij}}{\sum_{(i,j)\in B} \tau_{ij}}$.
μ ist ein Schätzwert für $\mu^* \in [-|V|C, +|V|C]$, welcher per Intervallschachtelung verbessert wird. $(C = \max c_{ij})$
Es wird ein Digraph $\overline{D}$ erstellt mit $l_{ij} = c_{ij} - \mu\tau_{ij}$ für alle $(i,j) \in A$.

- Enthält $\overline{D}$ einen negativen Kreis, ist μ zu groß.
- Enthält $\overline{D}$ nur positive Kreise, ist μ zu klein.
- Enthält $\overline{D}$ einen Kreis der Länge 0, minimiert dieser die Zielfunktion.

Das Verfahren kann auch abgebrochen werden, wenn die Intervallgrenzen weniger als $\frac{1}{n^2T^2}$ auseinander liegen. $(T = \max \tau_{ij})$
Laufzeit: $\mathcal{O}(VA \log(CVT))$

5 Das Zuordnungsproblem

5.1 Anwendungen und Grundlagen

Perfektes-Matching-Problem in bipartiten Graphen
Sei $K_{n,n}$ der vollständige bipartite Graph mit jeweils n Knoten in den zwei Teilmengen V_1 und V_2 mit $V_1 \cup V_2 = V$ und Kantengewichten c_{ij} für $i \in V_1$ und $j \in V_2$. Gesucht ist ein perfektes Matching ($\forall v \in V : |\delta(v)| = 1$) in $K_{n,n}$ mit minimalem Gesamtgewicht.

Das Problem wird mit Hilfe linearer Programmierung gelöst werden. Es lässt sich wie folgt als lineares Programm darstellen:

$$
\begin{aligned}
& \min \quad \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
(\mathrm{AP}) \qquad & \sum_{j=1}^{n} x_{ij} = 1, \text{ für } i = 1, 2, \ldots, n \\
& \sum_{i=1}^{n} x_{ij} = 1, \text{ für } j = 1, 2, \ldots, n \\
& x_{ij} \in \{0, 1\}, \text{ für } i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n
\end{aligned}
$$

Das duale Programm dazu ist:

$$
\begin{aligned}
(\mathrm{AP}_D) \quad & \max \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \\
& u_i + v_j \leq c_{ij}, \text{ für } 1, 2, \ldots, n,\ j = 1, 2, \ldots n.
\end{aligned}
$$

Satz. *Schwacher Dualitätssatz (am Beispiel)*
Wenn x zulässig für (AP) ist und (u, v) zulässig für (AP_D) sind, gilt:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \geq \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$$

Satz. *Satz vom komplenetären Schlupf (am Bsp.)*
Ein Paar von zulässigen Lösungen x für (AP) und (u, v) für (AP_D) sind genau dann optimal, wenn gilt

$$u_i + v_j < c_{ij} \Rightarrow x_{ij} = 0$$

bzw.

$$x_{ij} > 0 \Rightarrow u_i + v_j = c_{ij}$$

5.2 Die Ungarische Methode

HungarianMethod(n, C):

(1) Bestimme eine dual zulässige Startlösung $\left(u^0, v^0\right)$, setze $i = 0$.

(2) Konstruiere zu $\left(u^i, v^i\right)$ einen 0/1-Vektor x^i, so dass die Bedingungen vom komplementären Schlupf erfüllt sind.

(3) Beschreibt x^i eine Zuordnung, *Stop*(„Zuordnung optimal“)

(4) Berechne eine neue dual zulässige Lösung $\left(u^{i+1}, v^{i+1}\right)$, setze $i = i + 1$, gehe zu (2).

(Dies ist die grobe Idee des Algorithmus)

Definition 5.1. Folgende Definitionen werden für die Methode benötigt:

a) Für eine dual zulässige Lösung (u, v) heißt die Matrix $\overline{C} = (\overline{c}_{ij})$ mit $\overline{c}_{ij} = c_{ij} - u_i - v_j$ die **reduzierte Matrix**.

b) Sei $\overline{C}$ eine reduzierte Matrix. eine Menge $N \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ heißt Menge von **unabhängigen Nullen**, falls

(1) $\overline{c}_{ij} = 0$, für alle $(i,j) \in N$,

(2) $|\{j|\,(i,j) \in N\}| \leq 1$, für alle $i = 1,\ldots,n$,

(3) $|\{i|\,(i,j) \in N\}| \leq 1$, für alle $j = 1,\ldots,n$.

c) Eine **Überdeckung** einer reduzierten Matrix ist eine Menge von Zeilen- und Spaltenindizes, so dass die zugehörigen Zeilen und Spalten sämtliche Nullelemente von $\overline{C}$ enthalten.

Satz 5.2. *Sei $\overline{C}$ eine reduzierte Matrix. Die maximale Kardinalität einer Menge von unabhängigen Nullen ist gleich der minimalen Kardinalität einer Überdeckung.*
(In einem bipartiten Graphen ist die maximale Kardinalität eines Matchings gleich der minimalen Kardinalität einer Kantenüberdeckung mit Knoten.)

5.2.1 Bestimmung einer Startlösung

Start(n, C):

(1) Berechne Zeilenminima $u_i = \min\{\overline{c}_{ij}|j = 1,\ldots,n\}$, für alle $i = 1,\ldots,n$.

(2) Setze $\overline{c}_{ij} = c_{ij} - u_i$, für alle $i = 1,\ldots,n$, $j = 1,\ldots,n$.

(3) Berechne Spaltenminima $v_j = \min\{\overline{c}_{ij}|i = 1,\ldots,n\}$, für alle $j = 1,\ldots,n$.

(4) Setze $\overline{c}_{ij} = \overline{c}_{ij} - v_j$, für alle $i = 1,\ldots,n$, $j = 1,\ldots,n$.

Wir gehen nun spaltenweise vor und zeichnen die Null mit dem niedrigsten Zeilenindex als unabhängige Null aus.

5.2.2 Bestimmung einer minimalen Überdeckung

MinimusCover$(n, \overline{C})$:

(1) Alle Zeilen und Spalten von $\overline{C}$ seien unmarkiert und nicht überprüft. Sind n unabhängige Nullen vorhanden gehe zu (5).

(2) Alle Zeilen ohne unabhängige Null erhalten die Markierung „(−)“.

(3) Sei i eine markierte, noch nicht überprüfte Zeile.
Gibt es keine solche Zeile, so gehe zu (4).

(3.1) Erkläre i für überprüft.

(3.2) Sei j eine unmarkierte Spalte, die in Zeile i eine Null enthält.
Gibt es keine solche Spalte, gehe zu (3).

(3.3) Markiere j mit „(i)“.

(3.4) Besitzt j schon eine unabhängige Null gehe zu (3.2).
Andernfalls kann durch Verfolgen der Markierungen ausgehend von j eine alternierende Kette von abhängigen und unabhänigen Nullen konstruiert werden. Ändere die Klassifizierung dieser Nullen und gehe zu (1).

(4) Sei j eine markierte, noch nicht überprüfte Spalte.
Gibt es keine solche Spalte, so gehe zu (5).

(4.1) Erkläre j für überprüft.

(4.2) Markiere die Zeile, in der die unabhängige Null von Spalte j steht, mit „(j)“.

(4.3) Gehe zu (3).

(5) Die Maximalzahl unabhängiger Nullen ist berechnet. Die zugehörige minimale Überdeckung ergibt sich durch Zeilen, die nicht markiert sind und Spalten, die markiert sind.

5.2.3 Korrektur der Duallösung

I bezeichnet die Indizes der Zeilen der minimalen Überdeckung, J die der Spalten.

DualUpdate$\left(n, \overline{C}, I, J\right)$:

(1) Setze $\delta = \min\{\overline{c}_{ij} | i \notin I \wedge j \notin J\}$.

(2) Setze $u_i = \begin{cases} u_i + \delta, & \text{für } i \notin I, \\ u_i & \text{sonst,} \end{cases} \qquad v_j = \begin{cases} v_j - \delta, & \text{für } j \notin J, \\ v_j & \text{sonst.} \end{cases}$

(3) Setze $\overline{c}_{ij} = \begin{cases} \overline{c}_{ij} - \delta, & \text{für } i \notin I, j \notin J, \\ \overline{c}_{ij} + \delta, & \text{für } i \in I, j \in J, \\ \overline{c}_{ij}, & \text{sonst.} \end{cases}$

Satz 5.3. *Die Ungarische Methode löst Zuordnungsprobleme in der Zeit* $\mathcal{O}\left(n^3\right)$.

5.3 Ein dualer Algorithmus

Im Folgenden werden diese Notationen genutzt:

- $w_{ij} = c_{ij} - u_i - v_j$
- $I = \{1, 2, \ldots, n\}$ ist die Menge der Zeilenknoten
- $J = \{1, 2, \ldots, n\}$ ist die Menge der Spaltenknoten
- $G(I, J)$ ist der vollständige, bipartite Graph mit Kantenmenge $\{(i, j) | i \in I, j \in J\}$.

Definition 5.4. Sei F ein Subgraph von $G(I, J)$.

a) F heißt dual zulässig, falls es einen dual zulässigen Vektor (u, v) gibt, so dass $w_{ij} = 0$, für alle $(i, j) \in F$.

b) F heißt primal zulässig, falls es einen primal zulässigen Vektor x gibt, so dass $x_{ij} = 0$, für alle $(i, j) \in F$.

Lemma 5.5. *Ist F ein primal und dual zulässiger Subgraph von $G(I, J)$, dann sind die zugehörigen Vektoren (u, v) und x optimal für (AP) bzw. (AP_D).*

Definition. Sei F ein Subgraph von $G(I, J)$. Ein Knoten hat **Valenz** d (bzgl. F), falls sein Knotengrad in F gleich d ist.

Definition 5.6. Ein Subgraph F von $G(I, J)$ heißt **Superwald** mit einer Menge von Wurzeln aus J, falls gilt:

- F ist ein aufspannender Wald
- Jede Komponente von F enthält genau eine Wurzel
- Jeder Knoten aus J, der keine Wurzel ist, hat Valenz 2.

Definition. Ein Superwald F lässt sich in den **Überschuss-Wald** F^S und den **Defizit-Wald** F^D partitionieren, mit
F^S = Vereinigung der Komponenten von F, deren Wurzeln Valenz ≥ 2 haben,
F^D = Vereinigung der Komponenten von F, deren Wurzeln Valenz 0 oder 1 haben.

Lemma 5.7. *Ist F dual zulässiger Superwald mit $F^S = \emptyset$, dann ist der zugehörige Vektor (u, v) eine Optimallösung von (AP_D) und F enthält eine optimale Zuordnung.*

AKP(C)*:*

(1) Bestimme einen dual zulässigen Superwald F_0 mit Dualvariablen $\left(u^0, v^0\right)$ und setze $i = 0$.

(2) Falls F_i dual zulässig und F_i^S leer ist, Stop*(„F_i enthält eine optimale Zuordnung“).*

(3) Iteration i:

(3.1) Bestimme eine geeignete Kante (g, h).

(3.2) Konstruiere mit Hilfe von (g, h) einen neuen Superwald F_{i+1}.

(3.3) Berechne neue Dualvariablen $\left(u^{i+1}, v^{i+1}\right)$.

(3.4) Setze $i = i + 1$ und gehe zu (2).

5.3.1 Bestimmung einer Startlösung

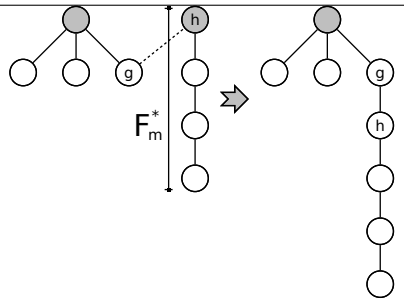
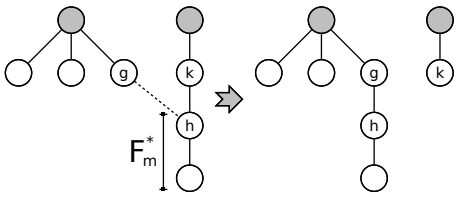
Bestimme u_i und v_j als Zeilen- und Spaltenminima. Erstelle Subgraph $F = \{(i,j)\,|w_{ij} = 0\}$. Entferne Kanten und zeichne Knoten aus J als Wurzeln aus, bis F ein Subgraph ist.

5.3.2 Korrektur des Superwaldes

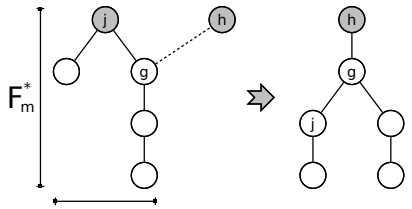
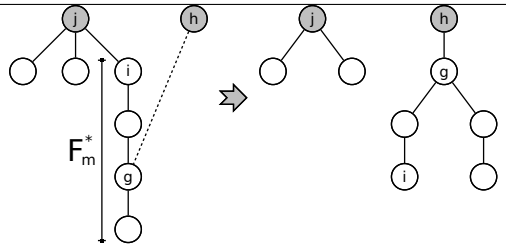
Wähle die Kante (g,h) mit kleinstem w_{ij}, die einen I-Knoten im Überschuss-Wald und einen J-Knoten im Defizit-Wald verbindet und füge sie dem Superwald hinzu:

$$\delta = w_{gh} = \min\left\{w_{ij} | i \in I \cap F_m^S, j \in J \cap F_m^D\right\}$$

Die Superwald-Eigenschaft muss wiederhergestellt werden. Es können 4 Fälle auftreten.
In Fall 1 und 2 wird der Überschuss-Wald größer:

Fall 1: h ist eine Wurzel mit Valenz 1	**Fall 2:** h ist keine Wurzel (und hat Vater k)
	
F_m^* = Komponente von F_m, die h enthält $F_{m+1}^D = F_m^D \setminus F_m^*$ $F_{m+1}^S = F_m^S \cup F_m^* \cup \{(g,h)\}$	F_m^* = Komponente von $F_m^D \setminus \{(k,h)\}$, die h enthält $F_{m+1}^D = F_m^D \setminus (F_m^* \cup \{(k,h)\})$ $F_{m+1}^S = F_m^S \cup (F_m^* \cup \{(g,h)\})$

In Fall 3 und 4 wird der Überschuss-Wald kleiner:

Fall 3: h ist isolierte Wurzel und die Wurzel j der Komponente von g hat Valenz 2	**Fall 4:** h ist isolierte Wurzel und die Wurzel j der Komponente von g hat Valenz >2
	
F_m^* = Komponente von F_m^S, die g enthält $F_{m+1}^D = F_m^D \cup F_m^* \cup \{(g,h)\}$ $F_{m+1}^S = F_m^S \setminus F_m^*$	F_m^* = Komponente von $F_m^S \setminus \{(j,i)\}$, die g enthält, mit i als Sohn von j im Ast der g enthält $F_{m+1}^D = F_m^D \cup F_m^* \cup \{(g,h)\}$ $F_{m+1}^S = F_m^S \setminus (F_m^* \cup \{(j,i)\})$

5.3.3 Korrektur der Dualvariablen

F_m^* ist anschaulich die Komponente, die „umgehängt“ wurde.
Ist Fall 1 oder 2 eingetreten, gilt:

$$u_i = \begin{cases} u_i - \delta, & \text{falls } i \in F_m^* \\ u_i & \text{sonst,} \end{cases}$$

$$v_j = \begin{cases} v_j + \delta, & \text{falls } j \in F_m^* \\ v_j & \text{sonst.} \end{cases}$$

In Fall 3 und 4 sind + und − vertauscht.

Lemma 5.8. *Falls die Teilwälder F_m^D und F_m^S dual zulässig sind, dann gilt $\delta_{m+1} \geq \delta_m$ und $\delta_{m+1} \geq$-δ_m.*

Lemma 5.9. *Falls F_m^D und F_m^S dual zulässig sind, dann auch F_{m+1}^D und F_{m+1}^S.*

Satz 5.10. *Der AKP-Algorithmus arbeitet korrekt und löst das Zuordnungsproblem mit Zeitkomplexität $\mathcal{O}\left(n^3\right)$.*

6 Maximale Flüsse und minimale Schnitte

Definition. Ein **Flussnetzwerk** ist ein Digraph $D = (V, A)$ mit Kantenkapazitäten $c(u, v) \geq 0$ für alle Kanten (u, v), einer **Quelle** $s \in V$ und einer **Senke** $t \in V$. D ist zusammenhängend und für jeden Knoten $v \in V$ existiert ein (s, v)-Weg so wie ein (v, t)-Weg.

Definition 6.1. Ein (s, t)**-Fluss** in D ist eine Funktion $f : V \times V \to \mathbb{R}$ mit den Eigenschaften

$f(u, v) \leq c(u, v)$, für alle $u, v \in V$ (Kapazitätsbeschränkung)

$f(u, v) = -f(u, v)$, für alle $u, v \in V$ (Antisymmetrie)

$\sum_{v \in V} f(u, v) = 0$, für alle $u \in V \setminus \{s, t\}$ (Flusserhaltung)

Die Zahl $f(u, v)$ heißt Netto-Fluss von u nach v. Der **Wert** von f ist $|f| := \sum_{v \in V} f(s, v)$.

Maximaler-(s, t)-Fluss-Problem
Gegeben sind ein Digraph $D = (V, A)$ mit Kantenkapazitäten und zwei Knoten $s, t \in V$. Zu bestimmen ist ein (s, t)-Fluss maximalen Werts.

Für $(u, v) \notin A$ setzen wir $c(u, v) = 0$.
Der **positive Nettofluss** in einen Knoten v ist definiert als

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v)$$

Der positive Nettofluss *aus* einem Knoten ist analog.
Für Knotenmengen X und Y gilt die Kurzschreibweise

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$$

Lemma 6.2. *Sei D ein Netzwerk und f ein Fluss.*

a) *Für $X \subseteq V$ gilt $f(X, X) = 0$.*

b) *Für $X, Y \subseteq V$ gilt $f(X, Y) = -f(Y, X)$.*

c) *Für $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$ gilt $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ und $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.*

6.1 Der Algorithmus von Ford und Fulkerson

Definition. Die **Restkapazität** ist definiert als $c_f(u, v) = c(u, v) - f(u, v)$. Das **reduzierte Netzwerk** $D_f = (V, A_f)$ mit $A_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$ enthält die möglichen, zusätzlichen Nettoflüsse.

Lemma 6.3. *Sei D ein Netzwerk mit (s, t)-Fluss f. D_f sei das zugehörige reduzierte Netzwerk und f' sei ein (s, t)-Fluss in D_f. Dann ist $\overline{f}$ definiert durch $\overline{f}(u, v) = f(u, v) + f'(u, v)$ ein (s, t)-Fluss in D mit Wert $|f| + |f'|$.*

Definition. Ein einfacher (s, t)-Weg in D_f heißt **augmentierender Weg** P mit **Restkapazität** $c_f(P) = \min\{c_f(u, v) \,|\, (u, v) \in P\}$

Definition. Ein (s, t)**-Schnitt** $(S : T)$ ist eine Partition von V in S und $T = V \setminus S$, wobei $s \in S$ und $t \in T$. Der **Nettofluss über dem Schnitt** $(S : T)$ ist definiert als $f(S, T)$. Seine Kapazität ist $c(S : T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$.

Minimaler-(s, t)-Schnitt-Problem
Gegeben sind ein Digraph $D = (V, A)$ mit Kantenkapazitäten und zwei Knoten $s, t \in V$. Zu bestimmen ist ein (s, t)-Schnitt minimaler Kapazität.

Lemma 6.4. *Seien f ein (s, t)-Fluss D und $(S : T)$ ein (s, t)-Schnitt. Dann gelten:*

a) *$f(S, T) = |f|$.*

b) *$|f| \leq c(S : T)$.*

Satz 6.5. ***(Max-Flow-Min-Cut-Theorem)***
Sei f ein (s, t)-Fluss in D. Die folgenden Aussagen sind äquivalent:

- *f ist maximaler Fluss.*
- *Das reduzierte Netzwerk enthält keinen augmentierenden Weg.*
- *Es gibt einen Schnitt $(S:T)$ mit $c(S:T) = |f|$.*

FordFulkerson(D, c, s, t):

(1) Für alle $(u, v) \in A$ setze $f[u, v] = f[v, u] = 0$.

(2) Konstruiere das reduzierte Netzwerk D_f.

(3) Falls kein augmentierender (s, t)-Weg existiert, *Stop*(„f ist maximal“), andernfalls sei P ein solcher Weg mit Restkapazität $c_f(P)$.

(4) Für jede Kante (u, v) des Weges P setze $f[u, v] = f[u, v] + c_f(P)$ und $f[v, u] = -f[u, v]$.

(5) Gehe zu (2).

Laufzeit: $\mathcal{O}(A|f^*|)$ -> nicht poly!
Ein minimaler Schnitt wird gleich mitgeliefert. Eine Seite sind alle Knoten, zu denen kein augmentierender Weg mehr gefunden werden kann.

Korollar 6.6. *Wenn alle Kapazitäten ganzzahlig sind, dann existiert ein ganzzahliger maximaler Fluss.*

Satz 6.7. *Sei f ein (s, t)-Fluss in D. Dann gibt es eine Familie von (s, t)-Wegen $\mathcal{P}$ und (gerichteten) Kreisen $\mathcal{C}$ in D mit Gewichten $w_P, P \in \mathcal{P}$, und $w_C, C \in \mathcal{C}$, so dass*

$$f(u, v) = \sum_{P \in \mathcal{P}} \sum_{(u,v) \in P} w_P + \sum_{C \in \mathcal{C}} \sum_{(u,v) \in C} w_C$$

6.2 Der Satz von Menger

(Dieses Kapitel zeigt, wie toll man einige Dinge mit Flüssen beweisen kann und *ergibt* ohne die Beweise leider nicht viel Sinn.)

Satz 6.8. *Seien $D = (V, A)$ ein gerichteter Graph und s, t zwei Knoten in V. Für $k \geq 1$ gibt es k kantendisjunkte (s, t)-Wege genau dann, wenn es auch nach Entfernen von $k - 1$ beliebigen Kanten noch einen Weg von s nach t gibt.*

Satz 6.9. *Seien $D = (V, A)$ ein Digraph und s, t zwei nicht benachbarte Knoten in V. Für $k \geq 1$ gibt es k intern knotendisjunkte (s, t)-Wege genau dann, wenn es auch nach Entfernen von $k - 1$ beliebigen Knoten noch einen Weg von s nach t gibt.*

6.3 Der Algorithmus von Edmonds und Karp

EdmondsKarp(D, c, s, t):

(1) Für alle $(u, v) \in A$ setze $f[u, v] = f[v, u] = 0$.

(2) Konstruiere das reduzierte Netzwerk D_f.

(3) Falls kein augmentierender (s, t)-Weg existiert, *Stop*(„f ist maximal“), andernfalls sei P ein solcher Weg *mit möglichst wenigen Kanten* und Restkapazität $c_f(P)$.

(4) Für jede Kante (u, v) des Weges P setze $f[u, v] = f[u, v] + c_f(P)$ und $f[v, u] = -f[u, v]$. Gehe zu (2).

Laufzeit: $\mathcal{O}(VA^2)$
$\delta_f(u, v)$ ist die Kürzeste-Wege-Distanz von u nach v im reduzierten Netzwerk, gemessen in Anzahl der Kanten.

Lemma 6.10. *Die Edmonds-Karp-Variante werde auf das NetzwerkD $= (V, A)$ mit Quelle s und Senke t angewendet. Dann nehmen für jeden Knoten $v \in V \setminus \{s, t\}$ die Distanzen $\delta_f(s, v)$ bei jeder Augmentierung nicht ab.*

Satz 6.11. *Die Edmonds-Karp-Variante führt $\mathcal{O}(VA)$ Augmentierungen durch.*

6.4 Die Skalierungsvariante von Ahuja und Orlin

ScalingMaxFlow(D, c, s, t):

(1) Setze $C = \max\{c(u,v) \mid (u,v) \in A\}$.

(2) Beginne mit dem (s,t)-Fluss $f \equiv 0$.

(3) Setze $K = 2^{\lfloor \log_2 C \rfloor}$.

(4) Solange $K \geq 1$:

(4.1) So lange es einen augmentierenden Weg mit Restkapazität $\geq k$ gibt, augmentiere f mit Hilfe dieses Weges.

(4.2) Setzte$K = K/2$.

(5) Der Fluss f ist ein maximaler Fluss.

Laufzeit: $\mathcal{O}\left(A^2 \log C\right)$, kann zu $\mathcal{O}\left(VA \log C\right)$ verbessert werden.

Lemma 6.12. *Vor der Ausführung von Schritt (4.1) ist die Kapazität eines minimalen Schnitts im reduzierten Netzwerk höchstens* $2 \cdot K \cdot |A|$.

Lemma 6.13. *Für festes K erfolgen $\mathcal{O}(A)$ Augmentierungen in Schritt (4.1).*

Satz 6.14. *Der Algorithmus ScalingMaxFlow hat die Zeitkomplexität $\mathcal{O}\left(A^2 \log C\right)$.*

6.5 Der Preflow-Push-Algorithmus

Definition 6.15. Ein Präfluss in einem Netzwerk D ist eine Funktion $f : V \times V \to \mathbb{R}$ mit den Eigenschaften

- $f(u,v) \leq c(u,v)$, für alle $u, v \in V$,
- $f(u,v) = -f(u,v)$, für alle $u, v \in V$,
- $f(V,u) \geq 0$, für alle $u \in V \setminus \{s\}$.

Der **Überschuss** von Knoten u ist $e(u) = f(V,u)$.
Ein Knoten $u \in V \setminus \{s,t\}$ heißt **aktiv**, falls $e(u) > 0$.

(Ein Präfluss ist also ein Fluss, der statt der Flusserhaltung „Flow-In = Flow-Out“ nur die Bedingung „Flow-In $\geq$ Flow-Out“ erfüllen muss.)

Definition 6.16. Sei D ein Netzwerk und f ein Präfluss. Eine Funktion $h : V \to \mathbb{N}_0$ heißt **Höhenfunktion**, falls

- $h(s) = |V|$,
- $h(t) = 0$,
- $h(u) \leq h(v) + 1$ für jeder Kante $(u,v), u \neq s$, aus dem reduzierten Netzwerk D_f.

Lemma 6.17. *Sei D ein Netzwerk, f ein Präfluss und h eine Höhenfunktion. Falls für zwei Knoten $u, v \in V$ $h(u) > h(v)$ gilt, dann ist(u,v) keine Kante des reduzierten Netzwerks.*

Push(u,v):

(1) Falls u aktiv ist, $h[u] = h[v] + 1$und $c_f(u,v) > 0$, dann:

(1.1) $\Delta = \min\{e[u], c_f(u,v)\}$.

(1.2) $f[u,v] = f[u,v] + \Delta$, $f[v,u] = -f[u,v]$, $e[u] = e[u] - \Delta$, $e[v] = e[v] + \Delta$.

Eine Push-Operation ist **saturierend**, wenn danach $c_f(u,v) = 0$ gilt, sonst **nicht saturierend**.

Lift(u):

(1) Falls u aktiv und es keine Kante $(u,v) \in A_f$ gibt mit $h[u] > h[v]$, dann:

(1.1) $h[u] = 1 + \min\{h[v] \mid (u,v) \in A_f\}$

PreflowPush(D, c, s, t):

(1) Für alle $v \in V$ setze $h[v] = 0$ und $e[v] = 0$.

(2) Für jede Kante $(u, v) \in A$ setze $f[u, v] = 0$ und $f[v, u] = 0$.

(3) Setze $h[s] = |V|$.

(4) Für jeden Knoten v mit $(s, v) \in A$ setze $f[s, v] = c(s, v)$, $f[v, s] = -c(s, v)$ und $e[v] = c(s, v)$.

(5) Solange eine anwendbare Push- oder Lift-Operation existiert, führe eine solche aus.

Laufzeit: $\mathcal{O}\left(V^2 A\right)$

Lemma 6.18. *Sei $D = (V, A)$ ein Netzwerk mit Quelle s und Senke t, f ein Präfluss und h eine Höhenfunktion. Wenn u ein aktiver Knoten ist, dann ist Lift(u) oder Push(u, v) für ein $v \in V$ anwendbar.*

Lemma 6.19. *Während der Ausführung von Preflow-Push nehmen die Höhen der Knoten nicht ab. Ist für einen Knoten u Lift(u) anwendbar, dann steigt die Höhe von u um mindestens 1.*

Lemma 6.20. *Die Funktion h behält während der Ausführung der Algorithmus die Eigenschaften einer Höhenfunktion.*

Lemma 6.21. *Es gibt keinen Weg von s nach t im reduzierten Netzwerk.*

Satz 6.22. *Nach Terminierung des Algorithmus ist ein maximaler (s, t)-Fluss bestimmt.*

Lemma 6.23. *Sei f ein Präfluss in $D = (V, A)$. Es gilt: Für jeden aktiven Knoten u gibt es einen (einfachen) Weg von u nach s im reduzierten Netzwerk D_f.*

Lemma 6.24. *Im Verlauf des Algorithmus gilt immer $h(u) \leq 2 \cdot |V| - 1$, für alle $u \in V$.*

Korollar 6.25. *Für jeden Knoten wird höchstens $(2 \cdot |V| - 1)$-mal eine Lift-Operation durchgeführt. Die Gesamtzahl aller Lift-Operationen ist höchstens $(2 \cdot |V| - 1)(|V| - 2) < 2 \cdot |V|^2$.*

Lemma 6.26. *Die Zahl der saturierenden Push-Operationen ist höchstens $2 \cdot |V| \cdot |A|$.*

Lemma 6.27. *Die Zahl der nichtsaturierenden Push-Operationen ist höchstens $4|V|^2 \cdot (|V| + |A|)$.*

Satz 6.28. *Die Zahl der Basis-Operationen bei der Ausführung des Preflow-Push-Algorithmus beträgt$\mathcal{O}\left(V^2 A\right)$.*

Korollar 6.29. *Der Preflow-Push-Algorithmus hat die Zeitkomplexität $\mathcal{O}\left(V^2 A\right)$.*

FIFO-Preflow-Push Hier werden Knoten in einer Queue abgearbeitet, die mit den Nachbarn von L initialisiert werden. Das abarbeiten funktioniert mit der *Examine*(u)-Operation:

Examine(u):

(1) Falls u aktiv ist, dann:

(1.1) Solange möglich, führe Operationen *Push*(u, v) aus. Wird v dadurch zu einem Überschussknoten, füge v am Ende von L ein.

(1.2) Ist *Lift*(u) anwendbar, führe die Operation aus und setze u ans Ende von L, andernfalls entferne u aus L.

Laufzeit: $\mathcal{O}\left(V^3\right)$

Highest-Label-Preflow-Push Bevorzuge Knoten mit dem höchsten h-Wert. Laufzeit: $\mathcal{O}\left(V^2\sqrt{A}\right)$

6.6 Untere Schranken und Knotenkapazitäten

6.6.1 Untere Schranken

Wenn ein **Mindestfluss** gesucht ist, könnte es die Bedingung $l(u, v) \leq f(u, v) \leq c(u, v)$ geben. Dazu transformieren wir das Problem in $D = (V, A)$ wie folgt:

$$
\begin{aligned}
D^* &= (V^*, A^*) \\
V^* &= V \cup \{s^*, t^*\} \\
A^* &= A \cup \{(s^*, v) \,|v \in V\} \cup \{(v, t^*) \,|v \in V\} \cup \{(t, s)\} \\
c^*(t, s) &= \infty \\
c^*(u, v) &= c(u, v) - l(u, v), \text{ für } (u, v) \in A \\
c^*(s^*, u) &= \sum_{(v,u)\in A} l(v, u), \text{ für } u \in V \\
c^*(u, t^*) &= \sum_{(u,v)\in A} l(u, v), \text{ für } u \in V
\end{aligned}
$$

In D existiert ein zulässiger Fluss genau dann, wenn der maximale (s^*, t^*)-Fluss in D^* den Wert $f^* = \sum_{(u,v)\in A} l(u, v)$ hat. Es gilt $f(u, v) = f^*(u, v) + l(u, v)$ ist ein zulässiger Fluss. Er kann als Startfluss für die oben genannten Max-Flow-Algorithmen genutzt werden.

6.6.2 Knotenkapazitäten

Alle Knoten werden in zwei Knoten geteilt, die mit einer Kante verbunden werden, die die Knotenkapatzität als Kapazität hat.

6.7 Anwendungen des Max-Flow-Min-Cut-Theorems

Hier könnten einige Anwendungen stehen, tun sie aber nicht.

Satz 6.30. *Es gilt das* ***Min-Flow-Max-Cut-Theorem***

$$
\min_{(s,t)\text{-}Fluss\ f} |f| = \max_{(s,t)\text{-}Schnitt\ (S:T)} \sum_{u\in S, v\in T} l(u, v) - \sum_{u\in S, v\in T} c(v, u).
$$

7 Schnitte in ungerichteten Graphen

Minimaler-Schnitt-Problem
Gegeben ist ein zusammenhängender ungerichteter Graph $G = (V, E)$ mit positiven Kantengewichten $c_e, e \in E$. Zu bestimmen ist eine Knotenmenge $W \subseteq V, \emptyset \neq W \neq V$, so dass $c(\delta(W))$ minimal ist.

7.1 Der Algorithmus von Nagamochi und Ibaraki

Definition. Der Graph G_{vw} entsteht durch **Identifikation** der beiden Knoten v und w, was bedeutet, dass sie durch einen neuen Knoten x ersetzt werden.

Lemma 7.1. *Die Schnitte in G_{vw} korrespondieren zu den Schnitten von G, die v und w nicht trennen.*

Lemma 7.2. *Seien $p, q, r \in V$. Es gilt $f(G : p, q) \geq \min\{f(G : p, r), f(G : q, r)\}$.*

Definition. Sei $v_1, v_2, \ldots, v_n$ eine Anordnung der Knoten von G und sei $V_i = \{v_1, \ldots, v_i\}, 1 \leq i \leq n$. Die Anordnung heißt **legal**, falls $c(\delta(V_{i-1}) \cap \delta(v_i)) \geq c(\delta(V_{i-1}) \cap \delta(v_j))$, für alle $2 \leq i \leq j \leq n$.
Man startet also mit irgendeinem Knoten und wählt als nächsten Knoten einen aus, dessen Summe der Kapazitäten der Kanten zu bereits gewählten Knoten maximal ist.

Satz 7.3. *Wenn $v_1, v_2, \ldots, v_n$ eine legale Ordnung für G ist, dann ist $\delta(v_n)$ ein minimaler (v_n, v_{n-1})-Schnitt in G.*

NagamochiIbaraki(G, c)*:*

(1) Setze $U = +\infty$, C undefiniert, $k = n = |V|$, $G^k = G$.

(2) Solange $k \geq 2$:

(2.1) Bestimme eine legale Ordnung $v_1, \ldots, v_k$ von G^k.

(2.2) Falls $c(\delta(v_k)) < U$, setze $U = c(\delta(v_k))$ und $C = \delta(v_k)$.

(2.3) Identifiziere die Knoten v_{k-1} und v_k.

(2.4) Setze $G^{k-1} = G^k_{v_{k-1}v_k}$ und $k = k - 1$.

(3) U ist Kapazität eines minimalen Schnitts in G und ein zugehöriger Schnitt lässt sich aus C konstruieren.

Laufzeit: $\mathcal{O}(V^3)$ oder $\mathcal{O}(V^2 + VE \log V)$

7.2 Der Algorithmus von Karger

Dies ist ein **randomisierter** Algorithmus, er liefert also mit einer gewissen Chance ein falsches Ergebnis.

Karger(G, c)*:*

(1) Solange G mehr als zwei Knoten enthält:

(1.1) Wähle eine Kante f von G mit Wahrscheinlichkeit $c_f / \sum_{e \in E} c_e$.

(1.2) Ersetze G durch G_{vw}, wobei $f = vw$.

(2) Gib den eindeutigen Schnitt von G aus.

Satz 7.4. *Sei $B \subseteq E$ ein minimaler Schnitt von G. Dann liefert der Algorithmus von Karger das Ergebnis B mit einer Wahrscheinlichkeit von mindestens $\frac{2}{n(n-1)}$.*

Korollar 7.5. *Sei B minimaler Schnitt von G und k eine positive ganze Zahl. Die Wahrscheinlichkeit, dass der Algorithmus von Karger bei $k \cdot n^2$ Aufrufen nicht mindestens einmal B als Ausgabe liefert, ist höchstens e^{-2k}.*

7.3 Minimale Schnitte zwischen allen Knotenpaaren

Alle-minimaler-Schnitte-Problem

Gegeben ist ein Graph $G = (V, E)$ mit positiven Kantengewichten und eine Knotenmenge $K \subseteq V$ (**Terminalknoten**). Zu bestimmen sind alle minimalen (u, v)-Schnitte für $u, v \in K$.

Hier könnte ihr Algorithmus stehen!

8 Flüsse mit minimalen Kosten

Betrachtet wird hier der Digraph $D = (V, A)$ mit **Kapazitäten** c_{uv}, **Kosten** w_{uv}, und einer Zahl b_u.
Knoten u mit $b_u > 0$ nennen wir **Quelle** mit **Vorrat**/**Angebot** b_u, bei $b_u < 0$ nennen wir sie **Senke** mit **Nachfrage** b_u, bei $b_u = 0$ heißen sie **Durchflussknoten** und müssen die Flusserhaltung erfüllen.

Minimale-Kosten-Fluss-Problem

Gegeben sind ein gerichteter Graph $D = (V, A)$ mit Kantenkapazitäten und -kosten sowie vorgegebenen Knotenbilanzen. Es ist ein Fluss gesucht, der alle Kapazitäten erfüllt, an den Knoten die Flussbilanzen realisiert und unter allen diesen Flussen minimale Kosten hat.

Als lineares Programm formuliert:

$$\min \sum_{(u,v) \in A} w_{uv} f_{uv}$$

$$\sum_{v:(u,v) \in A} f_{uv} - \sum_{v:(v,u) \in A} f_{vu} = b_u, \text{ für alle } u \in V$$

$$0 \leq f_{uv} \leq c_{uv}, \text{ für alle } (u, v) \in A$$

Annahmen:
Alle Daten sind ganzzahlig, es gibt so viel Angebot wie Nachfrage, es existiert ein zulässiger Fluss, alle Kosten und Kapazitäten sind nicht-negativ, es gibt keine antiparallelen Kanten (nur aus Notationsgründen).

Ein zulässiger Fluss lässt sich durch zusätzliche Knoten s^* und t^* und Kanten mit Kapazität von Angebot bzw. Nachfrage nachweisen.

8.1 Das klassische Transportproblem

(Unkapazitiertes) Transportproblem

Ein Produkt wird in m Anbieterorten produziert und in n Nachfrageorten benötigt. Am Anbieterort i liegt ein Lagervorrat a_i vor, am Nachfrageort j besteht der Bedarf b_j . (Es gelte $\sum a_i = \sum b_j$.) Der Transport einer Einheit des Produkts vom Ort i zum Ort j kostet w_{ij} . Zu bestimmen ist ein Transportplan, der alle Nachfragen erfüllt und minimale Kosten hat.

Als lineares Programm formuliert:

$$\begin{aligned} \min \sum_{i=1}^{m} \sum_{j=1}^{n} w_{ij} x_{ij} & \\ \sum_{j=1}^{n} x_{ij} = a_i & i = 1, \ldots, m \\ \sum_{i=1}^{m} x_{ij} = b_i & j = 1, \ldots, n \\ x_{ij} \geq 0 & i = 1, \ldots, m, j = 1, \ldots, n \end{aligned}$$

8.1.1 Die Nordwestecken-Regel

...liefert eine Startlösung.

NW(a, b):

(1) Setze $i = j = 1$ und $x_{11} = \min\{a_1, b_1\}$.

(2) Solange $i < m$ oder $j < n$:

(2.1) Setze $a_i = a_i - x_{ij}$ und $b_j = b_j - x_{ij}$.

(2.2) Falls $a_i = 0$, dann $i = i + 1$, sonst $j = j + 1$.

(2.3) Setze $x_{ij} = \min\{a_i, b_j\}$.

8.1.2 Die Stepping-Stone-Methode

Satz 8.1. *Eine Teilmenge E der Spalten von T ist genau dann linear unabhängig, wenn die zugehörige Kantenmenge in K_{mn} keinen Kreis enthält.*

Satz 8.2. *E ist eine Teilmenge von $m + n - 1$ linear unabhängigen Spalten von T genau dann, wenn der zugehörige Teilgraph von K_{mn} ein aufspannender Baum ist.*

Der Algorithmus sollte einfach auswendig gelernt werden. Probieren, studieren und so...

8.1.3 Das kapazitierte Transportproblem

Es existieren nun Kantenkapazitäten $0 \leq x_{ij} \leq c_{ij}$. Man fügt die Extraknoten $m + 1$ und $n + 1$ ein. Es gilt:

$$
\begin{aligned}
a_{m+1} = b_{n+1} &= \sum_{i=1}^{m} a_i \\
w_{i,n+1} = w_{m+1,j} &= M,\ i = 1, \ldots, m,\ j = 1, \ldots, n \\
w_{m+1,n+1} &= 0 \\
c_{i,n+1} &= a_i,\ i = 1, \ldots, m \\
c_{m+1,j} &= b_j,\ j = 1, \ldots, n \\
c_{m+1,n+1} &= \sum_{i=1}^{m} a_i
\end{aligned}
$$

Triviale Startlösung: Alle liefern an $n+1$, alle kriegen von $m+1$.
Das Originaleproblem hat eine Lösung, wenn das transformierte Problem eine Lösung hat, die die neuen Knoten nicht mit alten Knoten verbindet.

8.1.4 Varianten und Erweiterungen

<keine>

8.2 Optimalitätsbedingungen

Definition. Auch hier gibt es ein **reduziertes Netzwerk** $D_f = (V, A_f)$ mit Restkapazitäten r_{uv} zum Fluss f. Für eine Kante $(u, v) \in A$ gilt:

$$
\begin{aligned}
f_{uv} < c_{uv} &\Rightarrow (u, v) \in A_f, \text{ mit Kosten } +w_{uv} \text{ und } r_{uv} = c_{uv} - f_{uv} \\
f_{uv} > 0 &\Rightarrow (v, u) \in A_f, \text{ mit Kosten } -w_{uv} \text{ und } r_{uv} = f_{uv}
\end{aligned}
$$

Satz 8.3. ***(Negativer-Kreis-Bedingung)*** *Ein zulässiger Fluss f^* ist genau dann optimal, wenn das reduzierte Netzwerk D_{f^*} keinen Kreis enthält, so dass die Summe der Kosten seiner Kanten negativ ist.*

Satz 8.4. ***(Reduzierte-Kosten-Bedingung)*** *Ein zulässiger Fluss f^* ist genau dann optimal, wenn es einen Vektor π gibt, so dass $w^{\pi}_{uv} \geq 0$, für alle $(u, v) \in A_{f^*}$.*

8.3 Negaive-Kreise-Verfahren

CycleCanceling(D, c, w, b):

(1) Bestimme einen zulässigen Fluss f.

(2) Generiere das reduzierte Netzwerk D_f.

(3) Enthält D_f keinen negativen Kreis, dann *Stop*(„f is kostenminimal“). Andernfalls sein C ein negativer Kreis.

(4) Bestimme die Restkapazität Δ von C und setze $f_{uv} = f_{uv} + \Delta$ für alle $(u, v) \in C$. Gehe zu (2).

Laufzeit: $\mathcal{O}\left(V^2 AUW\right)$ mit U und W

Satz 8.5. *Sind alle Daten c_{uv}, $(u, v) \in A$, und b_u, $u \in V$, ganzzahlig, dann existiert ein kostenminimaler Fluss mit ganzzahligen Werten.*

8.4 Kürzerste-Wege-Verfahren

Definition. Für einen **Pseudofluss** f gilt $0 \leq f_{uv} \leq c_{uv}$ für alle $(u, v) \in A$. $f = 0$ **ist** ein Pseudofluss.
Für $v \in V$ ist die Knotenbilanz $e(v) = b_v \sum_{(u,v) \in A} f_{uv} - \sum_{(v,u) \in A} f_{vu}$.
Wir nennen Knoten mit $e(v) > 0$ **Überschussknoten**, mit $e(v) < 0$ **Defizitknoten**, mit $e(v) = 0$ **balanciert**.

Lemma 8.6. *Seien f ein Pseudofluss und π ein Vektor von Knotenpotentialen, so dass die Optimalitätsbedingungen $w^{\pi}_{uv} = \overline{w}_{uv} - \pi_u + \pi_v \geq 0$, für alle $(u, v) \in A$, erfüllt sind. Für jeden Knoten $v \in V$ sei $\delta(s, v)$ die Länge eines kürzesten Weges in D_f (bezüglich der Kantenlänge w^{π}) von einem ausgezeichneten Knoten s zu v.*

a) *Der Pseudofluss f erfüllt auch die Optimaitätsbedingungen mit den Knotenpotentialen π' definiert durch $\pi'_v = \pi_v - \delta(s,v)$, für alle $v \in V$.*

b) *Die reduzierten Kosten $w_{uv}^{\pi'}$ sind Null für alle Kanten (u,v), die auf einem kürzesten Weg von s zu einem anderen Knoten liegen.*

Lemma 8.7. *Sei f ein Pseudofluss, der die Reduzierte-Kosten-Bedingung erfüllt. Der Pseudofluss f' werde durch Augmentierung entlang eines kürzesten Weges (von s nach l in D_f bezpglich der Kantenlängen w^π) erhalten. Dann erfüllt f' die Reduzierte-Kosten-Bedingung bzgl. π'.*

SuccessiveShortestPaths(D, c, w, b):

(1) Setze $f = 0$, $\pi = 0$, und $e(u) = b_u$, für alle $u \in V$.

(2) Solange noch Überschussknoten existieren:

(2.1) Wähle einen Überschussknoten s und einen Defizitknoten t.

(2.2) Bestimme die Kürzeste-Wege-Distanzen $\delta(s,i)$ von s zu allen anderen Knoten i in D_f bezüglich der Kantenlängen w^π. Sei P der kürzeste Weg von s nach t.

(2.3) Setze $\pi_u = \pi_u - \delta(s,u)$, für alle $u \in V$.

(2.4) Augmentiere f entlang P um den Betrag $\min\{c_f(P), e(s) - e(t)\}$, wobei $c_f(P)$ die Restkapazität des Weges P ist.

(2.5) Bestimme die neuen reduzierten Kosten und das neue reduzierte Netzwerk.

Laufzeit: $\mathcal{O}(BV^2)$ oder $\mathcal{O}(BA \log V)$ mit B als Summe über alle Angebote.

SuccessiveShortestPaths2(D, c, w, b):

(1) Setze $f = 0$, $\pi = 0$, und $e(u) = b_u$, für alle $u \in V$.

(2) Solange ein Überschussknoten existiert:

(2.1) Wähle einen Überschussknoten s.

(2.2) Führe den Dijkstra-Algorithmus mit Startknoten s aus und stoppe ihn, sobald ein Defizitknoten permanent markiert wird. Wird kein Defizitknoten erreicht, *Stop*(„Problem unlösbar“).

(2.3) Sei t der erste permanent markierte Defizitknoten.

(2.4) Setze $\pi_u = \begin{cases} \pi_u - \delta(s,u) & u \text{ ist permanent markiert worden,} \\ \pi_u - \delta(s,t) & \text{sonst.} \end{cases}$

(2.5) Augmentiere f auf dem Weg von s nach t um den maximal möglichen Betrag.

(2.6) Bestimme die neuen reduzierten Kosten und das neue reduzierte Netzwerk.

Dieser Algorithmus funktioniert auch ohne die Annahme, dass alle Kürzeste-Wege-Distanzen existieren.

8.5 Der primale Simplex-Algorithmus

Omitted

Satz 8.8. —

8.6 Skalierung der Kapazitäten

Definition. In einer Δ-Skalierungs-Phase erfolgen jeweils Augmentierungen um mindestens den Betrag Δ. Das Δ-reduzierte Netzwerk enthält nur Kanten mit mindestens Restkapazität Δ und Quellen (Senken) mit mindestens Überschuss (Defizit) Δ.

Es gilt $U = \max\{\max\{c_{uv} | (u,v) \in A\}, \max\{b_u | b_u > 0\}, \max\{-b_u | b_u < 0\}\}$.

CapacityScaling(D, c, w, b):

(1) Setze $f = 0$, $\pi = 0$, $\Delta = 2^{\lfloor \log_2 U \rfloor}$.

(2) Solange $\Delta \geq 1$:

(2.1) Für alle Kanten $(u, v) \in A_f$ mit $r_{uv} \geq \Delta$ und $w^{\pi}_{uv} < 0$:

(2.1.1) Setze $f_{uv} = f_{uv} + r_{uv}$, $e(u) = e(u) - r_{uv}$ und $e(v) = e(v) + r_{uv}$.

(2.1.2) Setze $A_f = A_f \setminus \{(u, v)\} \cup \{(v, u)\}$ mit $r_{vu} = f_{uv}$

(2.2) Setze $S(\Delta) = \{v | e(v) \geq \Delta\}$ und $T(\Delta) = \{v | e(v) \leq -\Delta\}$

(2.3) Solange $S(\Delta) = \emptyset$ und $T(\Delta) = \emptyset$:

(2.3.1) Wähle $s \in S(\Delta)$ und $t \in T(\Delta)$.

(2.3.2) Berechne Kürzeste-Wege-Distanzen $\delta(s, v)$, für alle $v \in V$. Sei P der kürzeste (s, t)-Weg.

(2.3.3) Setze $\pi_v = \pi_v - \delta(s, v)$, für alle $v \in V$.

(2.3.4) Augmentiere entlang P um $\min\{c_f(P), e(s), -e(t)\}$.

(2.3.5) Ändere $S(\Delta)$, $T(\Delta)$ und D_f entsprechend.

(2.4) Setze $\Delta = \Delta/2$.

Laufzeit: $\mathcal{O}(AV^2 \log U)$ oder $\mathcal{O}(A^2 \log V \log U)$ mit Dijkstra.

Satz 8.9. *Die Laufzeit von* CapacityScaling *ist richtig.*

8.7 Minimale-Mittlere-Kreise-Verfahren

.....