# Eine graphentheoretische Herleitung und Implementierung des Netzwerk-Simplex-Algorithmus

Max Kanold

20. November 2018

# Inhaltsverzeichnis

# 1 Einführung

Das Simplex-Verfahren, zu welchem eine Einführung in [1, Chvátal, 2002] gefunden werden kann, löst Lineare Programme in der Praxis sehr schnell, obwohl die Worst-Case-Laufzeit nicht polynomiell ist. Jedes Netzwerkproblem lässt sich als Lineares Programm darstellen und somit mit dem Simplex-Verfahren lösen, durch die konkrete Struktur solcher Probleme genügt jedoch der von [2, Dantzig, 1951] und [3, Orden, 1956] vereinfachte Netzwerk-Simplex-Algorithmus. Auch für diesen gibt es exponentielle Instanzen (siehe Kapitel 4), in der Praxis wird er trotzdem vielfach verwendet.

TODO: Die Einführung wird ohnehin ausgebaut. Irgendwie erwähne ich auch unauffällig, dass in dieser Bachelorarbeit die Laufzeit des Algorithmus als Iterationsanzahl definiert ist, obwohl in der Praxis natürlich die Gesamtgeschwindigkeit zählt.

Sämtliche Bilder von Graphen im Verlauf dieser Bachelorarbeit sind nach folgender Legende zu lesen. Teilweise werden nur einzelne Werte angezeigt, Ausnahme ist die Kapazität, die nie ohne Flusswerte auftritt.

- b-Wert: schwarz im Knoten
- Fluss: blau auf Kante
- Kapazität: schwarz auf Kante, mit Schrägstrich (/) vom Fluss abgetrennt
- Kosten: rot auf Kante, falls notwendig mit Komma (,) abgetrennt

# 2 Der Netzwerk-Simplex-Algorithmus

Zunächst führen wir in Abschnitt 2.1 das Transportproblem und dessen Verallgemeinerung auf beschränkte Kapazitäten ein. Weiterhin geben wir die Einschränkungen an, auf denen insbesondere der Programmierteil dieser Bachelorarbeit fußt. Die Abschnitte 2.2 und 2.3 orientieren sich zuerst an [1, S. 291 ff.] für die Beschreibung des Netzwerk-Simplex-Algorithmus zur Lösung des Transportproblems, danach wird in Abschnitt 2.4 der Algorithmus anhand von [1, S. 353 ff.] auf den allgemeinen Fall erweitert.

Wir werden im Folgenden nur endliche Graphen betrachten. Außerdem sind alle Graphen einfach, das heißt, sie weisen weder mehrfache Kanten noch Schleifen auf. Der später eingeführte Residualgraph wird ebenfalls keine Schleifen besitzen, doppelte Kanten können unter Umständen vorkommen.

## 2.1 Min-Cost-Flow-Problem

**Definition 1.** Ein **Netzwerk** ist ein Tupel $(G, b, c, u)$ bestehend aus einem gerichteten Graphen $G = (V, E)$ und den Abbildungen $b\colon V \to \mathbb{R}$, $c\colon E \to \mathbb{R}$ sowie $u\colon E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. Wir bezeichnen $b$ als *b-Wert-Funktion*, $c$ als *Kostenfunktion* und $u$ als *Kapazitätsfunktion*.

**Anmerkung.** Ein ungerichteter Graph kann durch das Verwandeln jeder Kante $\{v, w\}$ in zwei Kanten $(v, w)$ und $(w, v)$ zu einem gerichteten modifiziert werden. Knoten mit positivem b-Wert werden als *Quellen*, solche mit negativem als *Senken* bezeichnet. Knoten mit neutralem b-Wert werden *Transitknoten* genannt.

**Definition 2.** Ein **maximaler Fluss** auf einem Netzwerk $(G = (V, E), b, c, u)$ ist eine Abbildung $f\colon E \to \mathbb{R}_{\geq 0}$, die folgende Eigenschaften erfüllt:

$$\begin{aligned} &\text{(i)} \quad \forall e \in E\colon \quad f(e) \leq u(e) \\ &\text{(ii)} \quad \forall v \in V\colon \quad \sum_{(w,v)\in E} f((w,v)) - \sum_{(v,w)\in E} f((v,w)) + b(v) = 0 \end{aligned} \tag{2.1}$$

Die *Kosten* von $f$ betragen $c(f) = \sum_{e\in E} f(e) \cdot c(e)$.

**Definition 3.** Sei $N = (G, b, c, u)$ ein Netzwerk. Als **Min-Cost-Flow-Problem** bezeichnen wir die Suche nach einem maximalen Fluss $f$ auf $N$ mit minimalen Kosten. Die vereinfachte Version mit unbeschränkter Kapazitätsfunktion, bei der $u(e) = \infty$ für alle $e \in E(G)$ gilt, nennen wir **Transportproblem**.

In dieser Bachelorarbeit wird angenommen, dass $b$ auf $\mathbb{Z}$ sowie $c$ und $u$ auf $\mathbb{N}_{\geq 0}$ abbilden, um Gleitkommazahlungenauigkeit beim Programmieren zu vermeiden. Durch eine entsprechende Skalierung des Problems können Funktionen nach $\mathbb{R}$ bzw. $\mathbb{R}_{\geq 0}$ hinreichend genau angenähert werden. Es wird davon ausgegangen, dass die b-Werte der Quellen und Senken ausgeglichen sind, also $\sum_{v \in V(G)} b(v) = 0$ gilt. Sollte die Summe über die Senken überwiegen, ist die Instanz unlösbar. Der umgekehrte Fall kann durch eine Dummy-Senke[1] abgefangen werden.

Da die Kostenfunktion auf nicht-negative Werte eingeschränkt ist, hat kein maximaler Fluss negative Kosten und unbeschränkte Instanzen sind ausgeschlossen. Unbeschränkte Kapazitäten können somit in der konkreten Implementierung durch $1 + 0,5 \cdot \sum_{v \in V} |b(v)|$ abgeschätzt werden, ohne dass die Lösungsmenge verändert wird. Alle Netzwerke werden als zusammenhängend angenommen, da die Zusammenhangskomponenten einer Instanz separat gelöst werden können. Der implementierte Algorithmus verlangt keinen Zusammenhang.

## 2.2 Zulässige Baumlösungen

Um die Funktionsweise des Netzwerk-Simplex-Algorithmus zu beschreiben, benötigen wir zunächst einige grundlegende Definitionen der Graphentheorie. Im Nachfolgenden sind die wichtigsten in Kürze aufgeführt, für eine vollständige Einführung sei der geneigte Leser auf [5] verwiesen.

**Definition 4.** Der einem gerichteten Graphen $G = (V, E)$ **zugrundeliegende ungerichtete Graph** $G' = (V, E')$ ist definiert durch:

$$\{v, w\} \in E' \iff (v, w) \in E \vee (w, v) \in E$$

**Anmerkung.** Dieser Definition zufolge bleibt der zugrundeliegende Graph einfach. Eine gängige Alternative entfernt nur die Orientierung aller Kanten, dies ist für unsere Zwecke jedoch nicht praktikabel.

**Definition 5.** Ein **Baum** $T$ ist ein ungerichteter, zusammenhängender und kreisfreier Graph. Ein **Wald** ist ein Graph, bei dem jede Zusammenhangskomponente ein Baum ist. Ein Teilgraph $T = (V', E')$ eines ungerichteten Graphen $G = (V, E)$ heißt **aufspannender Baum**, wenn T ein Baum und $V' = V$ ist.

**Anmerkung.** Bezeichnen wir einen gerichteten Graphen $G$ als Wald oder Baum, so bezieht sich das auf den $G$ zugrundeliegenden ungerichteten Graphen. Ein aufspannender Baum von $G$ ist ein Teilgraph $T$ von $G$, dessen zugrundeliegender ungerichteter Graph ein aufspannender Baum des $G$ zugrundeliegenden ungerichteten Graphen ist.

[1] Gemäß [4, S. 454] fügen wir $G$ eine zusätzliche Senke $s$ hinzu, die mit allen Quellen $q_i$ über eine Kante $e_i = (q_i, s)$ verbunden ist und einen b-Wert von $b(s) = -\sum_{v \in V(G)} b(v)$ zugewiesen bekommt. Für die neuen Kanten gilt $c(e_i) = 0$ und $u(e_i) = \infty$.

**Definition 6.** Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems. Ein aufspannender Baum $T$ von $G$ und ein maximaler Fluss $f$ auf $N$ bilden eine **zulässige Baumlösung** $(T, f)$, falls für alle Kanten $e \in E(G) \backslash E(T)$ außerhalb des Baumes $f(e) = 0$ gilt.

**Notation.** Sei $G = (V, E)$ ein Graph und $V' \subseteq V$ eine Teilmenge der Knoten. Der von V' *induzierte Teilgraph* $G[V'] = (V', E')$ mit $E' = \{(v, w) \in E \mid v \in V' \wedge w \in V'\}$ enthält die Knoten aus $V'$ und alle Kanten zwischen ihnen, die schon in $E$ vorhanden waren.

**Lemma 7.** *Jede zulässige Baumlösung $(T, f)$ ist eindeutig durch den aufspannenden Baum $T$ definiert.*

*Beweis.* Sei $(T, f)$ eine zulässige Baumlösung einer Instanz des Transportproblems. Wir führen eine Induktion über die Anzahl der Knoten des Baumes durch. Die zulässige Baumlösung zum leeren Baum ist offensichtlich eindeutig.

Für jedes Blatt $l \in V(T)$ sei $e_l \in E(T)$ die Kante in $T$ zwischen dem Knoten $l$ und seinem eindeutigen Nachbarknoten $k_l$. Für alle Blätter $l$ ist der Wert $f(e_l)$ nach Gleichung (2.1) eindeutig. Wir betrachten nun die eingeschränkte Knotenmenge $V' = \{v \in V(T) \mid v \text{ ist kein Blatt in } T\}$. Da jeder nichtleere Baum mindestens ein Blatt besitzt, ist $|V'| < |V(T)|$. Sei $T' = T[V'] \subsetneq T$ der durch $V'$ induzierte Teilgraph von $T$. Damit ist $T'$ ein Baum und $E(T')$ sind genau die Kanten, für die $f$ noch nicht bestimmt wurde.

Sei $N' = (G' = G[V'], b', c_{|E(G')}, u_{|E(G')})$ das auf $V'$ eingeschränkte Netzwerk $N$ mit der folgendermaßen angepassten b-Wert-Funktion $b'$, bei der nur die b-Werte von Blattnachbarn bezogen auf Blätter in $T$ angepasst werden:

$$\forall v' \in V': \quad b'(v') = b(v') + \sum_{\substack{l \in V(T) \backslash V': \\ (l, v') \in E(T)}} b(l) - \sum_{\substack{l \in V(T) \backslash V': \\ (v', l) \in E(T)}} b(l) \tag{2.2}$$

Wir nehmen per Induktion an, dass wir für den aufspannenden Baum $T'$ von $G'$ einen eindeutigen Fluss $f'$ erhalten, und setzen für alle Kanten $e \in E(T')$ den Fluss $f(e) := f'(e)$. Nach Gleichung (2.2) ist $f$ ein maximaler Fluss auf $N$, außerdem ist $f$ nach Konstruktion eindeutig. □

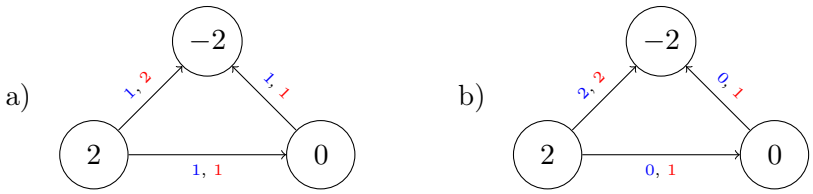


Abbildung 2.1: Links ein maximaler Fluss ohne zulässige Baumlösung, rechts ein maximaler Fluss mit mehreren zulässigen Baumlösungen.

Wie Abb. 2.1 a) veranschaulicht, existiert nicht zwingend ein Baum $T$ zu einem Fluss $f$; die Gegenrichtung von Lemma 7 gilt also nicht. Graph b) zeigt, dass auch bei Beschränkung auf maximale Flüsse mit zulässiger Baumlösung der Baum nicht eindeutig ist.

Wie wir in Korollar 11 zeigen werden, existiert für das Min-Cost-Flow-Problem immer eine Lösung, die auch eine zulässige Baumlösung ist. Die dem Algorithmus zugrundeliegende Idee ist, über die Bäume zulässiger Baumlösungen mit sinkenden Kosten zu iterieren. Der Übergang basiert dabei auf dem Augmentieren negativer Kreise, eine Methode, für die das Konzept des Residualnetzwerks hilfreich ist.

**Definition 8.** Sei $N = (G = (V, E), b, c, u)$ ein Netzwerk mit einem maximalen Fluss $f$. Die **Residualkante** $\bar{e}$ einer Kante $e = (v, w) \in E$ verläuft von $w$ nach $v$. Sei $\bar{E} = \{\bar{e} \mid e \in E\}$ die Menge aller Residualkanten. Die *Residualkapazität* $u_f \colon \bar{E} \to \mathbb{N}_{\geq 0}$ ist bestimmt durch $u_f(\bar{e}) = f(e)$, die *Residualkosten* $c_f \colon \bar{E} \to \mathbb{Z}$ durch $c_f(\bar{e}) = -c(e)$.

Das **Residualnetzwerk** $R$ ist ein Tupel $R_{N,f} = (\bar{G} = (V, E \amalg \bar{E}), \bar{f}, b, \bar{c}, \bar{u})$ mit dem *Residualgraph* genannten gerichteten Multigraphen $\bar{G}$ aus der bisherigen Knotenmenge und der disjunkten Vereinigung von Kanten und Residualkanten, dem maximalen Fluss $\bar{f}$ mit $\bar{f}(e) = f(e)$ für alle $e \in E$ und $\bar{f}(\bar{e}) = 0$ für alle $\bar{e} \in \bar{E}$, der b-Wert-Funktion $b$ wie gehabt, der Kostenfunktion $\bar{c} = c \amalg c_f \colon E \amalg \bar{E} \to \mathbb{Z}$ und der Kapazitätsfunktion $\bar{u} = u \amalg u_f \colon E \amalg \bar{E} \to \mathbb{N}_{\geq 0} \cup \{\infty\}$.

**Notation.** Mit $\varphi \colon E \leftrightarrow \bar{E}$ bezeichnen wir in einem Residualnetzwerk die kanonische Bijektion zwischen den Kanten und ihren Residualkanten. Außerdem seien für einen beliebigen Graph $G' = (V', E')$ die Kosten $c(G') := \sum_{e \in E'} c(e)$ als die Summe der Kosten der einzelnen Kanten definiert.

Wenn wir in einem Residualnetzwerk $R_{N,f}$ den Fluss einer Residualkante $\bar{e}$ um $0 \leq \delta \leq u_f(\bar{e})$ erhöhen, so wird in Wirklichkeit der Fluss $f(\varphi(\bar{e}))$ um $\delta$ reduziert. Nach obiger Definition ist sichergestellt, dass $f(\varphi(\bar{e})) - \delta \geq 0$.

Verändern wir den maximalen Fluss $f$ auf nur einer Kante, so ist die resultierende Funktion $f'$ kein maximaler Fluss mehr. Deswegen werden wir stattdessen gerichtete Kreise $C$ im Residualgraph $\bar{G}$ zu einer Abbildung $f'$ augmentieren, das heißt, wir erhöhen den Fluss auf allen Kanten $e \in E(C)$ um einen festen Betrag $\delta \in \mathbb{N}_{\geq 0}$, ohne dabei Kapazitätsschranken zu verletzen. Damit ist Gleichung (2.1) für maximale Flüsse weiterhin für alle Knoten $v \in V(C)$ erfüllt:

$$\begin{aligned} &\sum_{(w,v) \in E(G)} f'((w,v)) \quad - \sum_{(v,w) \in E(G)} f'((v,w)) \quad + b(v) \\ = &\sum_{(w,v) \in E(G)} f((w,v)) + \delta - \sum_{(v,w) \in E(G)} f((v,w)) - \delta + b(v) = 0 \end{aligned}$$

Also ist $f'$ ein maximaler Fluss. Seine Kosten betragen $c(f') = c(f) + \delta \cdot c(C)$.

**Lemma 9.** *Sei $N$ eine Instanz des Transportproblems, $R_{N,f}$ ihr Residualnetzwerk und $C$ ein gerichteter Kreis im Residualgraph $\bar{G}$ mit negativen Kosten. Dann ist der größte*

*Wert $\delta$, um den $C$ augmentiert werden kann, endlich, und nach der Augmentierung um $\delta$ zum neuen maximalen Fluss $f'$ existiert eine Residualkante $\bar{e} \in E(C)$, sodass die korrespondierende Kante einen Fluss von $f'(\varphi(\bar{e})) = 0$ besitzt.*

*Beweis.* Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems mit maximalen Fluss $f$ und $C$ ein negativer Kreis in $\bar{G}$. Wir setzen $\delta := \min_{e \in E(C)}\{\bar{u}(e) - \bar{f}(e)\}$. Da $c\colon E(G) \to \mathbb{N}_{\geq 0}$ in die natürlichen Zahlen abbildet, enthält jeder negative Kreis in $\bar{G}$ mindestens eine Residualkante $\bar{e}$. Damit ist $0 \leq \delta \leq u_f(\bar{e}) < \infty$.

Wir haben $\delta$ unter allen zulässigen Werten größtmöglich gewählt. Da alle Kanten $e \in E(G)$ unbeschränkte Kapazität haben und ihr Fluss $f(e)$ endlich ist, gibt es eine Residualkante $\bar{e}$ mit $u_f(\bar{e}) = \delta$. Nachdem wir $C$ um $\delta$ zu einem maximalen Fluss $f'$ augmentiert haben, gilt für die korrespondierende Kante $e := \varphi(\bar{e})$:

$$f'(e) = f(e) - \delta = f(e) - u_f(\bar{e}) = f(e) - f(e) = 0. \qquad \square$$

**Notation.** Sei $f$ ein maximaler Fluss für ein Netzwerk $(G = (V, E), b, c, u)$ und $H = (V' \subseteq V, E' \subseteq E)$ ein Teilgraph von $G$. Mit $H_f = (V', \{e \in E' \mid f(e) \neq 0\})$ bezeichnen wir den Graph aller durchflossenen Kanten von $H$.

Dank Lemma 9 wissen wir, dass nach maximaler Augmentierung eines negativen Kreises $C$ alle seine durchflossenen Kanten $C_f$ einen Wald bilden. Damit können wir nun zeigen, dass eine zulässige Baumlösung $(T, f)$ mit einem maximalen Fluss $f$ minimaler Kosten existiert.

**Theorem 10.** *Sei $N$ eine Instanz des Transportproblems mit einem maximalen Fluss $f$. Es existiert ein maximaler Fluss $\hat{f}$, sodass $c(\hat{f}) \leq c(f)$ ist und eine zulässige Baumlösung $(\hat{T}, \hat{f})$ existiert.*

*Beweis.* Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems mit maximalen Fluss $f$. Wir werden $f$ schrittweise zu einem maximalen Fluss $\hat{f}$ umwandeln, sodass $G_{\hat{f}}$ ein Wald ist. Für $\hat{f}$ lässt sich dann leicht eine zulässige Baumlösung finden. Wenn wir für die endlich vielen maximalen Zwischenflüsse $f = f_0, f_1, f_2, \ldots, f_n = \hat{f}$ sicherstellen, dass $c(f_{i+1}) \leq c(f_i)$ ist, gilt auch $c(\hat{f}) \leq c(f)$.

Betrachte einen maximalen Fluss $f_i$. Wenn $G_{f_i}$ ein Wald ist, setzen wir $\hat{f} := f_i$ und sind fertig. Ansonsten gibt es einen ungerichteten Kreis $C \subseteq G_{f_i}$. Betrachte die beiden dazugehörigen, gerichteten, kantendisjunkten Kreise $C_1$ und $C_2$ in $R_{N,f_i}$. Nach Definition 8 gilt $c(C_1) = -c(C_2)$. Wir werden einen der Kreise so augmentieren, dass $|E(G_{f_{i+1}})| < |E(G_{f_i})|$ ist.

Fall 1: $c(C_1) = 0$
Mindestens einer der beiden Kreise enthält eine Residualkante, sei dies o. B. d. A. $C_1$. Augmentiere nun $C_1$ analog zum Beweis von Lemma 9 größtmöglich zu einem maximalen Fluss $f_{i+1}$. Damit ist $C \nsubseteq G_{f_{i+1}}$ und $c(f_{i+1}) = c(f_i)$.

Fall 2: $c(C_1) \neq 0$
O. B. d. A. sei $c(C_1) < 0$. Nach Lemma 9 erhalten wir einen maximalen Fluss $f_{i+1}$, sodass $C \nsubseteq G_{f_{i+1}}$ und $c(f_{i+1}) = c(f_i) + \delta \cdot c(C_1) < c(f_i)$.

Damit ist $G_{f_{i+1}} \subsetneq G_{f_i}$ sowie $c(f_{i+1}) \leq c(f_i)$. Nach endlich vielen Iterationen erhalten wir somit ein kreisfreies $G_{f_j}$, dann sei $\hat{f} := f_j$. □

**Korollar 11.** *Für jede lösbare Instanz des Transportproblems existiert eine zulässige Baumlösung $(T, f)$, sodass der maximale Fluss $f$ minimale Kosten hat.* □

**Notation.** Sei $N = (G, b, c, u)$ ein Netzwerk mit maximalen Fluss $f$, $T$ ein aufspannender Baum von $G$, $R_{N,f} = (\bar{G}, \bar{f}, b, \bar{c}, \bar{u})$ das Residualnetzwerk und $e \in E(\bar{G}) \backslash E(T)$ eine weitere Kante. Mit $C_{T,e}$ bezeichnen wir den eindeutigen Teilgraph von $T \cup \{e\}$, dessen zugrundeliegender Graph ein Kreis ist, und mit $\bar{C}_{T,e}$ den $C_{T,e}$ eindeutig zugeordneten gerichteten Kreis im Residualgraph $\bar{G}$, der $e$ enthält.

## 2.3 Der grundlegende Algorithmus

Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems. Wie beim Simplex-Algorithmus gibt es beim Netzwerk-Simplex-Algorithmus zwei Phasen. In der ersten wird eine initiale zulässige Baumlösung $(T_0, f_0)$ auf $N$ erzeugt; die beiden etablierten Vorgehensweisen werden in Abschnitt 2.3.3 beschrieben. Die Problematik einer Instanz ohne Lösung wird ebenfalls dort behandelt.

In der zweiten Phase wird folgende Vorgehensweise iteriert: Betrachte Iteration $i$ mit zulässiger Baumlösung $(T_i, f_i)$. Wähle einen negativen Kreis $\bar{C}_{T,e}$, wobei $e \in E(G) \backslash E(T)$ eine Nichtbaumkante ist. Existiert kein solcher, beende den Algorithmus. Andernfalls augmentiere $\bar{C}_{T,e}$ größtmöglich zum maximalen Fluss $f_{i+1}$; dann existiert nach Lemma 9 eine Kante $e' \in E(C_{T,e})$ mit $e \neq e'$ und $f_{i+1}(e') = 0$. Die neue zulässige Baumlösung ist dann $(T_i - \{e'\} + \{e\}, f_{i+1})$. Die verschiedenen Möglichkeiten zur Wahl von $e'$ und $e$ werden in Abschnitt 2.3.1 und Abschnitt 2.3.2 näher beleuchtet.

**Theorem 12.** *Sei $N$ eine Instanz des Transportproblems mit einer zulässigen Baumlösung $(T, f)$. Existiert kein negativer Kreis $\bar{C}_{T,e}$, so ist $f$ eine optimale Lösung.*

Wir werden nun auf den Beweis von Theorem 12 hinarbeiten. Dieses sagt aus, dass der Algorithmus eine optimale Lösung des Transportproblems gefunden hat, wenn Phase 2 in Ermangelung einer geeigneten Kante $e$ beendet wird. Lemma 13 ist allgemeiner formuliert als notwendig, da wir in Abschnitt 2.4 darauf zurückgreifen werden.

**Notation.** Sei $N = (G, b, c, u)$ ein Netzwerk mit einem maximalen Fluss $f$, $T$ ein Baum von $G$ und $v, w \in V(T)$ zwei beliebige Knoten von $T$. Mit $v \xrightarrow{T} w$ bezeichnen wir den eindeutigen gerichteten Weg von $v$ nach $w$ im Residualgraph $\bar{G}$, der nur über Kanten $e \in E(T)$ oder deren Residualkanten $\varphi(e)$ verläuft. Insbesondere ist dieser Weg unabhängig von $f$ und für die Kosten gilt $c(w \xrightarrow{T} v) = -c(v \xrightarrow{T} w)$.

**Lemma 13.** *Sei $N = (G, b, c, u)$ ein Netzwerk mit einer zulässigen Baumlösung $(T, f)$ und $C$ ein ungerichteter Kreis in $G$, sodass für einen korrespondierenden gerichteten Kreis $\bar{C}$ im Residualgraph $\bar{G}$ gilt:*

*(i)* $c(\bar{C}) < 0$

*(ii)* $\forall e \in E(C) \backslash E(T)$: *die zu e korrespondierende Kante* $\bar{e} \in E(\bar{C})$ *darf vom Netzwerk-Simplex-Algorithmus bei zulässiger Baumlösung* $(T, f)$ *als eingehende Kante gewählt werden*

*Dann existiert eine vom Algorithmus wählbare Kante* $\bar{e} \in \bar{C}$*, sodass* $c(\bar{C}_{T,\bar{e}}) < 0$.

*Beweis.* Sei $N$ ein Netzwerk, $(T, f)$ eine zulässige Baumlösung sowie $C$ und $\bar{C}$ Kreise wie gefordert. Sei $E_{\bar{C},\neg T} \subseteq E(\bar{C})$ die Menge der zu den Nichtbaumkanten $E(C) \backslash E(T)$ korrespondierenden Kanten in $\bar{C}$ und $E_{\bar{C},T} = E(\bar{C}) \backslash E_{\bar{C},\neg T}$ dessen Komplement. Betrachten wir zunächst den Fall, dass $E_{\bar{C},T} = \emptyset$:

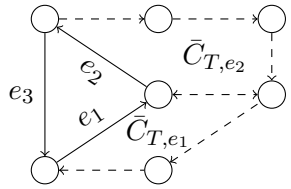


Abbildung 2.2: Die durchgezogenen Kanten bilden den negativen Kreis $\bar{C}$. Jede Kante $(v, w) \in E(\bar{C})$ wird über $w \xrightarrow{T} v$ zu einem Kreis ergänzt.

Nach Eigenschaft (ii) gibt es für jede Kante $e_i \in E_{\bar{C},\neg T} = E(\bar{C})$ einen Kreis $\bar{C}_{T,e_i}$. Wie Abb. 2.2 veranschaulicht, können wir einen Kreis $\bar{C}_{T,e_i}$ auch erzeugen, indem wir mit $\bar{C}$ beginnen, jede Kante $e_j = (v_j, w_j) \in E(\bar{C})$ mit $j \neq i$ durch den Weg $v_j \xrightarrow{T} w_j$ ersetzen und dabei in beide Richtungen begangene Kanten entfernen. Für ein $j$ verändern sich die Kosten dabei um $c(v_j \xrightarrow{T} w_j) - c(e_j) = -c(\bar{C}_{T,e_j})$, womit insgesamt gilt:

$$c(\bar{C}_{T,e_i}) = c(\bar{C}) - \sum_{j \neq i} c(\bar{C}_{T,e_j}) \quad \Leftrightarrow \quad \sum_{i=1}^{|E(\bar{C})|} c(\bar{C}_{T,e_i}) = c(\bar{C}) \tag{2.3}$$

Nach Gleichung (2.3) muss mindestens ein Kreis $\bar{C}_{T,e_i}$ negative Kosten besitzen, da $c(\bar{C}) < 0$ ist. Damit ist die Aussage gezeigt.

Kommen wir zum Fall $E_{\bar{C},T} \neq \emptyset$. Da $T$ ein Baum ist, ist auch $E_{\bar{C},\neg T} \neq \emptyset$. Sollte $|E_{\bar{C},\neg T}| = 1$ sein, ist $\bar{C}$ der gesuchte Kreis. Ansonsten werden wir $\bar{C}$ iterativ derart zu einem Kreis $\hat{C}$ verändern, dass $E_{\hat{C},\neg T} \subsetneq E_{\bar{C},\neg T}$ und $|E_{\hat{C},\neg T}| = 1$. Besitzt $\hat{C}$ negative Kosten, sind wir ebenfalls fertig, andernfalls werden wir bereits vorher einen negativen Kreis $\bar{C}_{T,e}$ mit $e \in E_{\bar{C},\neg T}$ gefunden haben.

Betrachte einen Iterationsschritt mit negativen Kreis $\bar{C}$, sodass $|E_{\bar{C},\neg T}| > 1$ gilt. Sei $e = (x, y) \in E_{\bar{C},\neg T}$ eine beliebige Nichtbaumkante. Wir betrachten nun den Kreis $\tilde{C} := \bar{C}_{T,e} = y \xrightarrow{T} x \cup \{(x, y)\}$ und den Kantenzug $W := x \xrightarrow{T} y \xrightarrow{\bar{C}-\{e\}} x$. Letzterer zerfällt – bereinigt um in beide Richtungen begangene Kanten – in kantendisjunkte Kreise $C_1, \ldots, C_w$ mit $c(W) = \sum_{i=1}^{c} c(C_i)$. O. B. d. A. sei der Knoten $x$ in $C_1$ enthalten.

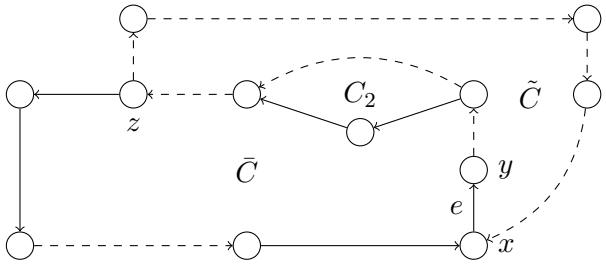


Abbildung 2.3: Gestrichelte Linien sind zum unvollständig dargestellten Baum $T$ assoziiert. Der neue Kreis $C_1$ entsteht hier, indem wir von $x$ beginnend entgegen $\tilde{C}$ bis $z$ gehen und dann dem bisherigen Kreisverlauf $\bar{C}$ folgen.

Sollten die Kosten $c(\tilde{C}) < 0$ negativ sein, ist $\tilde{C}$ nach Eigenschaft (ii) der gesuchte Kreis. Ist einer der Kreise $C_2, \ldots, C_c$ negativ, ersetzen wir $\bar{C}$ durch diesen Kreis. Andernfalls gilt für den Kreis $C_1$:

- $|E_{C_1,\neg T}| < |E_{\bar{C},\neg T}|$
- $c(C_1) = c(\bar{C}) - c(\tilde{C}) - \sum_{i=2}^{c} c(C_i) \leq c(\bar{C}) < 0$

Damit können wir $\bar{C} := C_1$ setzen. In beiden Fällen ist $|E_{\bar{C},\neg T}|$ im Vergleich zum Anfang der Iteration echt kleiner geworden. Wir iterieren weiter, bis $|E_{\bar{C},\neg T}| = 1$ ist; dann ist $\hat{C} := \bar{C}$ der gesuchte Kreis. □

Um unser gewünschtes Theorem 12 zu zeigen, genügt es, einen negativen Kreis gemäß Lemma 13 zu finden. Dafür werden wir den Begriff der Zirkulation einführen und den Zusammenhang zwischen maximalen Flüssen und Zirkulationen herstellen.

**Definition 14.** Sei $R_{N,f} = (\bar{G}, \bar{f}, b, \bar{c}, \bar{u})$ ein Residualnetzwerk. Eine **Zirkulation** auf $R_{N,f}$ ist eine Abbildung $z\colon E(\bar{G}) \to \mathbb{N}_{\geq 0}$, die folgende Eigenschaften erfüllt:

$$\text{(i)} \quad \forall e \in E(\bar{G})\colon \quad z(e) \leq \bar{u}(e) \tag{2.4}$$

$$\text{(ii)} \quad \forall v \in V(\bar{G})\colon \sum_{(w,v)\in E(\bar{G})} z((w,v)) - \sum_{(v,w)\in E(\bar{G})} z((v,w)) = 0 \tag{2.5}$$

Die *Kosten* von $z$ betragen $c(z) = \sum_{e\in E(\bar{G})} z(e) \cdot \bar{c}(e)$.

**Anmerkung.** Zirkulationen sind an sich für beliebige gerichtete Graphen mit einer Kapazitätsfunktion definierbar, für unsere Zwecke genügt die obige Version. Die Notation $\bar{G}_z$ für den Graph der durchflossenen Kanten überträgt sich.

**Lemma 15.** *Jede Zirkulation $z$ auf einem Residualnetzwerk $(\bar{G}, \bar{f}, b, \bar{c}, \bar{u})$ zerfällt in eine endliche Menge $\mathscr{Z} = \{z_1, \ldots, z_n\}$ von Zirkulationen, sodass für alle $z_i \in \mathscr{Z}$ die Graphen $\bar{G}_{z_i}$ paarweise disjunkte gerichtete Kreise sind, $z(e) = \sum_{i=1}^{n} z_i(e)$ für alle $e \in E(\bar{G})$ gilt und die Kosten $c(z) = \sum_{i=1}^{n} c(z_i)$ ebenfalls zerfallen.*

*Beweis.* Sei $R_{N,f} = (\bar{G}, \bar{f}, b, \bar{c}, \bar{u})$ ein Residualnetzwerk und $z$ eine Zirkulation. Wir führen eine Induktion über die Anzahl durchflossenener Kanten $|E(\bar{G}_z)|$ der Zirkulation durch. Ist $|E(\bar{G}_z)| = 0$, so erfüllt die leere Menge das Lemma.

Sei $|E(\bar{G}_z)| > 0$. Damit ist $\bar{G}_z$ nicht der leere Graph und nach Gleichung (2.5) existiert ein gerichteter Kreis $C \subseteq \bar{G}_z$. Wir setzen $\delta := \min_{e \in E(C)}\{z(e)\} > 0$ sowie $\tilde{e} = \arg\min_{e \in E(C)}\{z(e)\}$ und definieren eine Abbildung $z'$ wie folgt:

$$z'(e) = \begin{cases} z(e) - \delta & \text{falls } e \in E(C) \\ z(e) & \text{sonst} \end{cases}$$

Nach Konstruktion ist $z'$ eine Zirkulation und $|E(\bar{G}_{z'})| \leq |E(\bar{G}_z)\backslash\{\tilde{e}\}| < |E(\bar{G}_z)|$. Wir nehmen per Induktion an, dass wir für $z'$ eine Menge $\mathscr{Z}' := \{z'_1, \ldots, z'_n\}$ von Zirkulationen mit obigen Eigenschaften erhalten. Da $\tilde{e} \in E(C)$ nicht Teil des von $z'$ durchflossenen Graphen $\bar{G}_{z'}$ ist, ist $C$ disjunkt zu allen Kreisen $\bar{G}_{z'_i}$.

Wir definieren nun zu $C$ die Zirkulation $z_{n+1}$ mit $z_{n+1}(e) = \delta$ für alle Kreiskanten $e \in E(C)$ sowie $z_{n+1}(e) = 0$ sonst und setzen $\mathscr{Z} := \mathscr{Z}' \cup \{z_{n+1}\}$, womit für $z$ und $\mathscr{Z}$ nach Konstruktion gilt:

$$\forall e \in E(\bar{G})\backslash E(C)\colon \quad \sum_{z_i \in \mathscr{Z}} z_i(e) = \sum_{z'_i \in \mathscr{Z}'} z'_i(e) + z_{n+1}(e) = z'(e) + 0 = z(e)$$

$$\forall e \in E(C)\colon \quad \sum_{z_i \in \mathscr{Z}} z_i(e) = \sum_{z'_i \in \mathscr{Z}'} z'_i(e) + z_{n+1}(e) = z'(e) + \delta = z(e)$$

Für die Kosten gilt:

$$\sum_{z_i \in \mathscr{Z}} c(z_i) = \sum_{z'_i \in \mathscr{Z}'} c(z'_i) + c(z_{n+1}) = c(z') + \delta \cdot c(C) = c(z)$$

Damit ist die Aussage für alle Zirkulationen gezeigt. □

**Notation.** Seien $f$ und $\hat{f}$ zwei maximale Flüsse auf einem Netzwerk $(G, b, c, u)$. Mit $\hat{f} - f$ sei die folgendermaßen auf dem Residualnetzwerk $R_{N,f}$ definierte Abbildung $z\colon E(\bar{G}) \to \mathbb{N}_{\geq 0}$ gemeint:

$$z(e) = \begin{cases} \max\{\hat{f}(e) - f(e), 0\} & \text{falls } e \in E(G) \\ \max\{f(\varphi(e)) - \hat{f}(\varphi(e)), 0\} & \text{falls } e \in \varphi(E(G)) \end{cases}$$

**Lemma 16.** *Seien $f$ und $\hat{f}$ zwei maximale Flüsse auf einem Netzwerk $(G, b, c, u)$. Dann ist $z := \hat{f} - f$ eine Zirkulation auf $R_{N,f}$ und für die Kosten gilt $c(z) = c(\hat{f}) - c(f)$.*

*Beweis.* Sei $E := E(G)$ die Menge aller Kanten des Graphen und $\bar{E} := \varphi(E)$ die Menge ihrer Residualkanten in $\bar{G}$. Zunächst überprüfen wir, ob $z$ alle Eigenschaften aus Definition 14 erfüllt. Für alle Kanten $e \in E$ ist

$$z(e) = \max\{\hat{f}(e) - f(e), 0\} \leq \hat{f}(e) \leq u(e) = \bar{u}(e),$$

für Residualkanten $\bar{e} \in \bar{E}$ gilt

$$z(\bar{e}) = \max\{f(\varphi(\bar{e})) - \hat{f}(\varphi(\bar{e})), 0\} \leq f(\varphi(\bar{e})) \leq u_f(\bar{e}) \leq \bar{u}(\bar{e}).$$

Gleichung (2.4) ist folglich gegeben. Es bleibt zu zeigen, dass sich der Fluss von $z$ für alle Knoten $v \in V(\bar{G})$ gemäß Gleichung (2.5) ausgleicht:

$$\begin{aligned}
&\sum_{(w,v)\in E \amalg \bar{E}} z((w,v)) - \sum_{(v,w)\in E \amalg \bar{E}} z((v,w)) \\
&= \sum_{(w,v)\in E} \Big(\hat{f}((w,v)) - f((w,v))\Big) - \sum_{(v,w)\in E} \Big(\hat{f}((v,w)) - f((v,w))\Big) \\
&= b(v) - b(v) = 0
\end{aligned}$$

Die erste Gleichheit gilt gemäß Konstruktion von $z$, da ein theoretischer negativer Fluss auf einer Kante $e \in E(G)$ als positiver auf der Residualkante $\varphi(e)$ umgesetzt wird. Die zweite Umformung gilt gemäß Gleichung (2.1) aus Definition 2. Damit ist Gleichung (2.5) erfüllt und $z$ eine Zirkulation. Zu guter Letzt betrachten wir die Kosten von $z$:

$$\begin{aligned}
c(z) &= \sum_{e\in E \amalg \bar{E}} z(e) \cdot \bar{c}(e) \\
&= \sum_{e\in E} \max\{\hat{f}(e) - f(e), 0\} \cdot c(e) + \sum_{\bar{e}\in\bar{E}} \max\{f(\varphi(\bar{e})) - \hat{f}(\varphi(\bar{e})), 0\} \cdot c_f(\bar{e}) \\
&= \sum_{e\in E} \max\{\hat{f}(e) - f(e), 0\} \cdot c(e) + \sum_{\bar{e}\in\bar{E}} -\min\{\hat{f}(\varphi(\bar{e})) - f(\varphi(\bar{e})), 0\} \cdot (-c(\varphi(\bar{e}))) \\
&= \sum_{e\in E} \max\{\hat{f}(e) - f(e), 0\} \cdot c(e) + \sum_{e\in E} \min\{\hat{f}(e) - f(e), 0\} \cdot c(e) \\
&= \sum_{e\in E} (\hat{f}(e) - f(e)) \cdot c(e) = c(\hat{f}) - c(f)
\end{aligned}$$

□

Bevor wir nun Theorem 12 beweisen, zeigen wir noch, wie sich mithilfe einer Zirkulation der Beweis von Lemma 7 wesentlich vereinfachen lässt.

**Lemma 7.** *Jede zulässige Baumlösung $(T, f)$ ist eindeutig durch den aufspannenden Baum $T$ definiert.*

*Beweis.* Seien $(T, f)$ und $(T, f')$ zwei zulässige Baumlösungen zu demselben aufspannenden Baum $T$ des Netzwerkes $N = (G, b, c, u)$. Betrachte die Zirkulation $z := f' - f$ auf $R_{N,f}$. Da $G_f \cup G_{f'} \subseteq T$ ein Wald ist, gilt dies auch für $\bar{G}_z$. Damit gilt $z(\bar{e}) = 0$ für alle Kanten $\bar{e} \in E(\bar{G})$, womit schon $f = f'$ war. □

**Theorem 12.** *Sei $N$ eine Instanz des Transportproblems mit einer zulässigen Baumlösung $(T, f)$. Existiert kein negativer Kreis $\bar{C}_{T,e}$, so ist $f$ eine optimale Lösung.*

*Beweis.* Wir werden zeigen, dass ein negativer Kreis $\bar{C}_{T,e}$ existiert, falls die betrachtete zulässige Baumlösung $(T, f)$ nicht optimal ist. Dazu finden wir einen gerichteten negativen Kreis, auf den wir Lemma 13 anwenden können.

Sei $N = (G, b, c, u)$ ein Netzwerk mit einer optimalen zulässigen Baumlösung $(\hat{T}, \hat{f})$ und einer zulässigen Baumlösung $(T, f)$, sodass $c(f) > c(\hat{f})$. Dann ist $z := \hat{f} - f$ eine Zirkulation und nach Lemma 16 sind ihre Kosten $c(z) = c(\hat{f}) - c(f) < 0$ negativ. Des Weiteren zerfällt $z$ gemäß Lemma 15 in eine Menge $\mathscr{Z} = \{z_1, \ldots, z_n\}$ von Zirkulationen, sodass $c(z) = \sum_{i=1}^{n} c(z_i)$ gilt und $\bar{G}_{z_i}$ für alle $z_i \in \mathscr{Z}$ ein gerichteter Kreis ist.

Da $c(z) < 0$ ist, gibt es mindestens eine Zirkulation $z_j \in \mathscr{Z}$ mit negativen Kosten. Sei $C_j := \bar{G}_{z_j}$ der von ihr durchflossene Kreis und $e \in E(C_j)$ eine Kante davon. Ist $e$ eine Residualkante, so gilt $\hat{f}(\varphi(e)) - f(\varphi(e)) < 0$, woraus $f(e) > 0$ und damit $\varphi(e) \in E(T)$ folgt. Ansonsten ist $e \in E(G)$ und vom Algorithmus wählbar. Wir können also auf $C_j$ Lemma 13 anwenden, das Theorem folgt. □

**Korollar 17.** *Sei $N$ eine Instanz des Transportproblems mit einer initialen zulässigen Baumlösung. Determiniert der Netzwerk-Simplex-Algorithmus, so ist er korrekt.* □

Bislang haben wir nicht gezeigt, dass der Algorithmus immer terminiert. Dies wäre offensichtlich, wenn bei jeder Iteration von $(T, f)$ zu $(T', f')$ die Kosten sinken würden, also $c(f') < c(f)$ wäre. Es gibt jedoch sogenannte *degenerierte Iterationen*, in denen ein negativer Kreis $\bar{C}_{T,e}$ um $\delta = 0$ augmentiert wird. Entfernt der Algorithmus danach eine Kante $e' \in E(C_{T,e}) \backslash \{e\}$, verändert sich nur der Baum. Im nächsten Abschnitt lernen wir einfache Methode kennen, durch die der Algorithmus immer terminiert.

### 2.3.1 Degenerierte Iterationen

**Definition 18.** Wird in einer Iteration der Phase 2 des Netzwerk-Simplex-Algorithmus ein negativer Kreis $\bar{C}_{T,e}$ um $\delta = 0$ augmentiert, so bezeichnen wir diese als **degenerierte Iteration**.

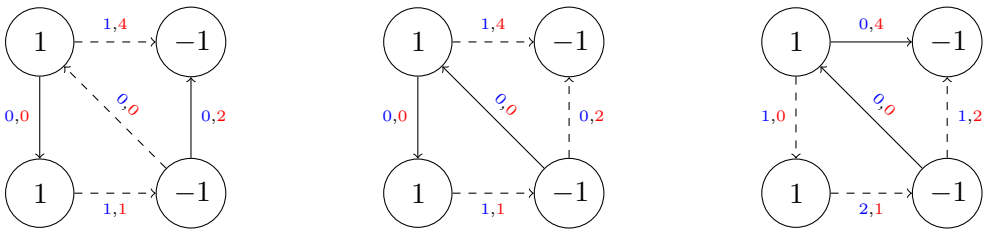


Abbildung 2.4: Die Kanten des Baumes $T_i$ sind gestrichelt dargestellt, der gewählte negative Kreis ist jeweils eindeutig. Die Startlösung kann erst nach einer degenerierten Iteration verbessert werden.

Degenerierte Iterationen entstehen, wenn bei einer zulässigen Baumlösung $(T, f)$ nicht alle Kanten von $T$ durchflossen sind, also $T \neq T_f$ gilt. $(T, f)$ wird dann auch als

*degeneriert* bezeichnet. In einer ungünstigen Konstellation der deterministischen Wahl der hinzugefügten Kante $e$ und entfernten Kante $e'$ kann es zum *Cycling* kommen, also zu einer Abfolge von Bäumen, die wiederholt iteriert werden. Dies tritt sehr selten auf, für ein konstruiertes Beispiel siehe [1, S. 303].

In [6] führte Cunningham 1976 eine Methode ein, mit der durch eine geschickte Wahl der entfernten Kante Cycling verhindert werden kann, ohne dass die Auswahl der hinzufügbaren Kanten eingeschränkt wird. Dafür benötigen wir folgende Definitionen:

**Definition 19.** Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems. Ein aufspannender Baum $T$ von $G$, ein maximaler Fluss $f$ auf $N$ und ein Wurzelknoten $r \in V(T) = V(G)$ bilden eine **stark zulässige Baumlösung** $(T, f)_r$, falls $(T, f)$ eine zulässige Baumlösung ist und zusätzlich jede Kante $e = (v, w) \in E(T)$ mit $f(e) = 0$ von der Wurzel wegführt, also $e$ im Weg $r \xrightarrow{T} w$ enthalten ist.

**Definition 20.** Sei $N$ ein Netzwerk mit einer stark zulässigen Baumlösung $(T, f)_r$ und $e \in E(G) \backslash E(T)$ eine weitere Kante. Der **Apex** von $C_{T,e}$ ist der eindeutige Knoten $p \in V(C_{T,e})$, der den kürzesten Weg zum Wurzelknoten $r$ aufweist.

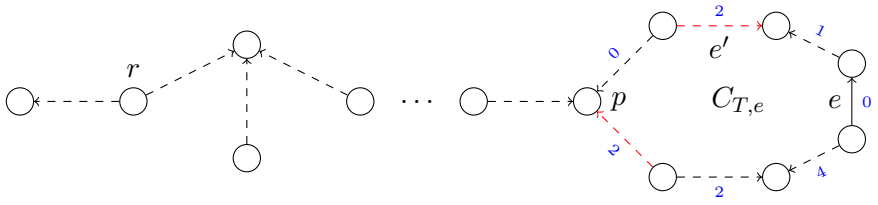


Abbildung 2.5: Grafik angelehnt an [7, S. 372]. Die gestrichelten Linien stellen den Baum $T$ mit Wurzelknoten $r$ dar. Die beiden blockierenden Kanten in $C_{T,e}$ sind rot gekennzeichnet, $p$ ist der Apex.

Für einen Kreis $C_{T,e}$ bezeichnen wir diejenigen Kanten, die verhindern, dass der Kreis um mehr als $\delta$ augmentiert werden kann, als *blockierende Kanten*. In Abb. 2.5 ist $\delta = 2$ und die blockierenden Kanten sind rot eingefärbt. Insbesondere wird die entfernte Kante immer unter diesen gewählt.

Die von Cunningham eingeführte *Regel der letzten blockierenden Kante* besagt nun folgendes: Mit dem Apex von $C_{T,e}$ beginnend gehe über alle Kanten $e \in E(C_{T,e})$ in der durch $e$ bestimmten Orientierung des Kreises und wähle die letzte blockierende Kante als die zu entfernende Kante aus. In Abb. 2.5 ist diese dementsprechend $e'$. Dank dieser Regel terminiert der Netzwerk-Simplex-Algorithmus nicht nur, er wird meistens sogar beschleunigt.

**Lemma 21** [6, S. 108 f.]**.** *Wird eine stark zulässige Baumlösung $(T, f)_r$ um einen Kreis $\bar{C}_{T,e}$ größtmöglich augmentiert und eine Kante gemäß der Regel der letzten blockierenden Kante entfernt, ist die entstehende Baumlösung $(T', f')$ wieder eine stark zulässige Baumlösung $(T', f')_r$.* □

**Theorem 22** [6, S. 108 f.]**.** *Der Netzwerk-Simplex-Algorithmus mit einer initialen stark zulässigen Baumlösung, einem beliebigen Pivotalgorithmus und der Regel der letzten blockierenden Kante terminiert.* □

Nach [7, S. 359] ist die Laufzeit des Netzwerk-Simplex-Algorithmus für bestimmte Pivotalgorithmen polynomiell in der Anzahl Knoten und Kanten sowie den höchsten Kosten einer Kante beschränkt. Für die in der Praxis verwendeten Pivotalgorithmen sind keine polynomiellen Schranken bekannt, teilweise wurden sogar exponentielle Instanzen gefunden, die auf *Stalling* basieren, also einer exponentiellen Anzahl degenerierter Iterationen. Mit diesen werden wir uns in Kapitel 4 näher befassen. Meine eigene, experimentelle Suche nach schlechten Instanzen findet sich in Kapitel 5. Zunächst vervollständigen wir den Algorithmus um die Wahl des negativen Kreises $\bar{C}_{T,e}$ und die Erzeugung einer initialen Baumlösung.

### 2.3.2 Pivotalgorithmen

Sei $N = (G, b, c, u)$ ein Netzwerk mit zulässiger Baumlösung $(T, f)$. Algorithmen, die aus der Menge $\bar{C}_T = \{\bar{C}_{T,e} \mid c(\bar{C}_{T,e}) < 0\}$ aller sinnvollen Iterationen eine auswählen, heißen *Pivotalgorithmen*. In der Praxis wird der Pivotalgorithmus nur auf einer Teilmenge von $\bar{C}_T$ ausgeführt, um Rechenzeit zu sparen. In dieser Bachelorarbeit werden nur drei naheliegende, einfache Pivotalgorithmen betrachtet.

#### Maximum Value

Der erste Ansatz ist es, den negativsten Kreis zu wählen, sprich

$$\tilde{C} := \operatorname*{arg\,min}_{\bar{C}_{T,e} \in \bar{C}_T} \{c(\bar{C}_{T,e})\}$$

Diesen Weg werden wir mit *MaxVal* bezeichnen.

#### Maximum Revenue

Die Kostenverringerung nach der Augmentierung beträgt $\delta_e \cdot c(\bar{C}_{T,e})$, ist also von $\delta_e$ abhängig. Der Pivotalgorithmus *MaxRev* maximiert diesen Wert:

$$\tilde{C} := \operatorname*{arg\,min}_{\bar{C}_{T,e} \in \bar{C}_T} \{\delta_e \cdot c(\bar{C}_{T,e})\}, \qquad \text{wobei } \delta_e := \min_{e' \in E(\bar{C}_{T,e})} \{\bar{u}(e') - \bar{f}(e')\}$$

Nach Lemma 9 ist jedes $\delta_e$ endlich. Sollten nur degenerierte Iterationen zur Auswahl stehen, ist $c(\tilde{C}) = 0$. In dem Fall wendet meine konkrete Implementierung MaxVal an; hier sind aber auch andere Strategien denkbar.

#### Random

Ein überraschend effektiver Ansatz ist es, $\tilde{C} \in \bar{C}_T$ zufällig zu wählen. Gerade für diesen mit *Random* bezeichneten Weg ist es schwierig, die erwartete Laufzeit zu bestimmen.

### 2.3.3 Initialisierung

Der letzte verbleibende Schritt, um einen vollständigen Algorithmus zur Lösung des Transportproblems zu erlangen, ist das Finden einer initialen, stark zulässigen Baumlösung $(T_0, f_0)_r$.

Hierfür wenden wir einen Trick an: Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems. Wir fügen dem Netzwerk einen zusätzlichen Transitknoten $a$ mit $b(a) = 0$ hinzu, der gemeinhin als künstlicher *(artificial)* Knoten bezeichnet wird. Für alle Quellen $v \in V(G)\colon b(v) > 0$ ergänzen wir eine künstliche Kante $(v, a)$, für alle Senken und Transitknoten $a \neq w \in V(G)\colon b(w) \leq 0$ eine künstliche Kante $(a, w)$. Sei $G'$ der entstehende Graph.

Sämtliche künstlichen Kanten $e_v$ haben eine unbegrenzte Kapazität, die Kosten sind ebenfalls unendlich und können mit $c(e_v) = |V(G)| \cdot \max_{e \in E(G)}\{c(e)\} + 1$ abgeschätzt werden, da jeder Weg in $G$ geringere Kosten aufweist. Leicht finden wir nun die stark zulässige Baumlösung $(T_0, f_0)_a$, wobei $T_0 = (V(G), \{e_v \mid v \in V(G)\})$ ist. Nach Lemma 7 ist $f_0$ eindeutig. Diese Art der Initialisierung werden wir im Folgenden mit *HC* für *High-Cost-Initialisierung* abkürzen.

**Lemma 23.** *Sei $N = (G, b, c, u)$ eine Instanz des Transportproblems und $(T, f)$ die Lösung des Netzwerk-Simplex-Algorithmus mit* HC *auf dem erweiterten Graphen $G'$. Die Instanz $N$ ist genau dann lösbar, wenn $G'_f$ keine künstlichen Kanten enthält.*

*Beweis.*

„⇒“ Sei $\hat{f}$ ein maximaler Fluss von $N$ und $(T, f)$ die Lösung des Algorithmus. $\hat{f}$ ist auch ein maximaler Fluss des Netzwerkes $(G', b, c, u)$; damit gilt nach Theorem 10 und Korollar 17 $c(f) \leq c(\hat{f}) < \infty$. Da die Kosten einer künstlichen Kante unendlich sind, kann keine davon in $G'_f$ enthalten sein.

„⇐“ Sei $(T, f)$ die Lösung des Algorithmus. Wenn $G'_f$ keine künstlichen Kanten enthält, ist $f_{|E(G)}$ ein maximaler Fluss von $N$, womit diese Instanz wiederum lösbar ist. □

Sollen Instanzen mit vielen Knoten oder hohen Kosten der teuersten Kante gelöst werden, so kann es passieren, dass die Kosten der künstlichen Kanten den darstellbaren Bereich der gängigen Datentypen sprengen. Unter anderem aus diesem Grund gibt es die *Low-Cost-Initialisierung*, bei uns kurz *LC*.

Bei dieser wird die gegebene Instanz $N = (G, b, c, u)$ des Transportproblems zunächst zu $N' = (G, b, c', u)$ abgewandelt. Dabei ist $c'$ die Nullfunktion, also $c(e) = 0$ für jede Kante $e \in E(G)$. Für $N'$ verwenden wir den Netzwerk-Simplex-Algorithmus mit *HC* und erhalten eine stark zulässige Baumlösung $(T', f')_a$, wobei die Kosten einer künstlichen Kante $c(e_v) = |V(G)| \cdot 0 + 1 = 1$ betragen.

Offensichtlicherweise ist $N$ genau dann lösbar, wenn $N'$ lösbar ist. Wenn $c(f') > 0$ ist, so ist $N$ nicht lösbar und wir sind fertig. Andernfalls iterieren wir $(T', f')_a$ degeneriert, bis $a$ ein Blatt von $T'$ ist. Dies wird in Abschnitt 3.3.3 detailliert erläutert.

Damit bekommen wir die zulässige Baumlösung $(T = T' - \{a\}, f = f'_{|E(G)})$ für $N'$ und damit für $N$. Als neuen Wurzelknoten können wir den eindeutigen Nachbar $k_a$ von

$a$ in $T'$ wählen; damit erhalten wir die stark zulässige Baumlösung $(T, f)_{k_a}$. Nun kann eine optimale Lösung des Transportproblems für $N$ berechnet werden.

## 2.4 Erweiterung auf beschränkte Kapazitäten

Bisher haben wir uns auf unbeschränkte Kapazitäten beschränkt. Wir werden nun den Netzwerk-Simplex-Algorithmus auf das allgemeinere Min-Cost-Flow-Problem erweitern. Die Funktionsweise bleibt dieselbe, nur die betrachteten zulässigen Baumlösungen verändern sich leicht:

**Definition 24.** Sei $N = (G, b, c, u)$ ein Netzwerk. Ein aufspannender Baum $T$ von $G$ und ein maximaler Fluss $f$ auf $N$ bilden eine **zulässige Baumlösung** $(T, f)$, wenn für alle Kanten $e \in E(G) \backslash E(T)$ außerhalb des Baums $f(e) = 0$ oder $f(e) = u(e)$ gilt.

**Anmerkung.** Kanten, die maximal durchflossen sind, bezeichnen wir als *saturiert*.

**Definition 25.** Sei $N = (G, b, c, u)$ ein Netzwerk. Ein aufspannender Baum $T$ von $G$, ein maximaler Fluss $f$ auf $N$ und ein Wurzelknoten $r \in V(T) = V(G)$ bilden eine **stark zulässige Baumlösung** $(T, f)_r$, wenn $(T, f)$ eine zulässige Baumlösung ist und folgendes gilt:

$$\forall e = (v, w) \in E(T): \quad (f(e) = 0 \Rightarrow e \in E(r \xrightarrow{T} w)) \quad \wedge$$
$$(f(e) = u(e) \Rightarrow e \in E(w \xrightarrow{T} r))$$

**Anmerkung.** An dieser Stelle sollten sämtliche Kanten $e \in E(G)$ mit $u(e) = 0$ aus dem Graphen entfernt werden; sie sind für das Problem ohnehin irrelevant.


Abbildung 2.6: Derselbe, durch gestrichelte Linien dargestellte Baum mit unterschiedlichen maximalen Flüssen. Der Wurzelknoten ist die Senke.

Im Falle beschränkter Kapazitäten erlauben wir für eine zulässige Baumlösung $(T, f)$ auch Fluss außerhalb von $T$, insofern er die Kapazität der Kante voll ausnutzt. Abb. 2.6 veranschaulicht, warum dies notwendig ist. Solche saturierten Kanten $e \in E(G) \backslash E(T)$ sind als eingehende Kante keine sinnvolle Wahl mehr, stattdessen kann der Algorithmus über den Kreis $\bar{C}_{T,\bar{e}}$ der Residualkante $\bar{e}$ augmentieren, sofern er negativ ist. Für stark zulässige Baumlösungen müssen nun zusätzlich alle saturierten Kanten des Baumes zum Wurzelknoten hinführen.

Abb. 2.6 veranschaulicht, dass die in Lemma 7 bewiesene Eindeutigkeit von $f$ nun nicht mehr gilt. Wir haben diese Eigenschaft bei der Initialisierung genutzt. Der Leser

kann sich an dieser Stelle davon überzeugen, dass diese trotzdem komplett identisch durchführbar ist.

Die in Abschnitt 2.3.1 beschriebene Regel der letzten blockierenden Kante wurde in [6] ohnehin für den allgemeinen Fall eingeführt und lässt sich einfach übertragen. Wir werden jetzt Lemma 9, Theoreme 10 und 12 und Korollar 11 auf das Min-Cost-Flow-Problem erweitern.

**Lemma 26.** *Sei $N$ ein Netzwerk, $R_{N,f}$ sein Residualnetzwerk und $C$ ein gerichteter Kreis im Residualgraph $\bar{G}$ mit negativen Kosten. Dann ist der größte Wert $\delta$, um den $C$ augmentiert werden kann, endlich, und nach der Augmentierung um $\delta$ zum neuen maximalen Fluss $f'$ existiert eine Residualkante $\bar{e} \in E(C)$, sodass die korrespondierende Kante einen Fluss von $f'(\varphi(e)) = 0$ besitzt, oder eine Kante $e \in E(C) \cap E(G)$ mit $f'(e) = u(e)$.*

*Beweis.* Sei $N = (G, b, c, u)$ ein Netzwerk mit maximalen Fluss $f$ und $C$ ein negativer Kreis in $\bar{G}$. Sei $\tilde{e} := \arg\min_{e \in E(C)}\{\bar{u}(e) - \bar{f}(e)\}$ und $\delta := \bar{u}(\tilde{e}) - \bar{f}(\tilde{e})$. Analog zum Beweis von Lemma 9 ist $\delta$ endlich.

Ist $\tilde{e}$ eine Residualkante, so besitzt $e := \varphi(e)$ einen Fluss von

$$f'(e) = f(e) - \delta = f(e) - u_f(\bar{e}) = f(e) - f(e) = 0.$$

Andernfalls ist $\tilde{e} \in E(C) \cap E(G)$ mit einem Fluss von

$$f'(\tilde{e}) = f(\tilde{e}) + \delta = f(\tilde{e}) + \bar{u}(\tilde{e}) - \bar{f}(\tilde{e}) = \bar{u}(\tilde{e}) = u(\tilde{e}). \qquad \square$$

**Notation.** Sei $f$ ein maximaler Fluss für ein Netzwerk $(G = (V, E), b, c, u)$ und $H = (V' \subseteq V, E' \subseteq E)$ ein Teilgraph. Mit $H^f = H_f \backslash \{e \in E' \mid f(e) = u(e)\}$ bezeichnen wir den Graph aller durchflossenen, aber nicht saturierten Kanten.

**Theorem 27.** *Sei $N$ ein Netzwerk mit einem maximalen Fluss $f$. Es existiert ein maximaler Fluss $\hat{f}$, sodass $c(\hat{f}) \leq c(f)$ ist und eine zulässige Baumlösung $(\hat{T}, \hat{f})$ existiert.*

*Beweis.* Sei $N = (G, b, c, u)$ ein Netzwerk mit maximalen Fluss $f$. Diesmal werden wir $f$ zu einem maximalen Fluss $\hat{f}$ umwandeln, sodass $G^{\hat{f}}$ ein Wald ist. Der Beweis verläuft nun komplett analog zum Beweis von Theorem 10, statt Lemma 9 benutzen wir Lemma 26. $\square$

**Korollar 28.** *Für jede lösbare Instanz des Min-Cost-Flow-Problems existiert eine zulässige Baumlösung $(T, f)$, sodass der maximale Fluss $f$ minimale Kosten hat.* $\square$

**Theorem 29.** *Sei $N$ ein Netzwerk mit einer zulässigen Baumlösung $(T, f)$. Existiert kein negativer Kreis $\bar{C}_{T,e}$, so ist $f$ eine optimale Lösung.*

*Beweis.* Wir werden wiederum zeigen, dass ein negativer Kreis $\bar{C}_{T,e}$ existiert, wenn die betrachtete Baumlösung nicht optimal ist.

Sei $N = (G, b, c, u)$ ein Netzwerk mit einer optimalen zulässigen Baumlösung $(\hat{T}, \hat{f})$ und einer zulässigen Baumlösung $(T, f)$, sodass $c(f) > c(\hat{f})$ ist. Wieder betrachten wir

die Zirkulation $z := \hat{f} - f$, ihre Zerlegung gemäß Lemma 15 in eine Menge $\mathscr{Z}$ von Zirkulationen über gerichteten Kreisen und eine Zirkulation $z_j \in \mathscr{Z}$ mit negativen Kosten sowie den ihr zugehörigen negativen Kreis $C_j = \bar{G}_{z_j}$.

Sei $e \in E(C_j)$ eine Kante dieses Kreises. Ist $e$ eine Residualkante, so gilt wie gehabt $\hat{f}(\varphi(e)) - f(\varphi(e)) < 0$, woraus $f(e) > 0$ folgt. Damit ist $\varphi(e) \in E(T)$ oder $\varphi(e)$ saturiert. Andernfalls ist $e \in E(G)$ und $f(e) < \hat{f}(e)$, womit $e$ nicht saturiert sein kann. Die Bedingungen für Lemma 13 sind erneut gegeben. □

**Korollar 30.** *Der Netzwerk-Simplex-Algorithmus mit einer Initialisierung nach Abschnitt 2.3.3, einem beliebigen Pivotalgorithmus und gemäß Abschnitt 2.3.1 auf stark zulässige Baumlösungen beschränkt löst das Min-Cost-Flow-Problem korrekt.* □

# 3 Implementierung

Im Programmierteil dieser Bachelorarbeit habe ich den Netzwerk-Simplex-Algorithmus wie in Kapitel 2 beschrieben implementiert. Der Code ist in C++ geschrieben und hält sich an den C++11-Standard. Es folgt eine vollständige Übersicht der Codebestandteile, für das Gesamtkonstrukt kann die angehängte CD konsultiert werden. Im Folgenden wurden an einigen Stellen für ein besseres Layout Variablennamen angepasst, Kommentare gekürzt und Funktionsdeklarationen weggelassen.

## 3.1 Netzwerke

Zur Umsetzung der Graphenstruktur habe ich eine Klasse `class Network` geschrieben, die auf `struct Node` sowie `class Edge` basiert und den Fluss händelt. Damit kann `Network` beispielsweise sicherstellen, dass nie Kanten entfernt werden, die aktuell durchflossen sind.

### 3.1.1 Knoten

`struct Node` hat außer einem Konstruktor keinerlei Funktionen, weswegen es nicht in einer eigenen Klasse ausgelagert, sondern in `network.h` enthalten ist.

```
struct Node {
  size_t id;
  intmax_t b_value;
  //defined by (the other) nodeID
  //always incoming and outgoing due to residual edges
  std::set<size_t> neighbours;

  Node (size_t _id, intmax_t _b_value)
    : id(_id), b_value(_b_value) {};
};
```

Ein Knoten wird eindeutig durch seine `size_t id` identifiziert, weitere Eigenschaften sind sein b-Wert aus $\mathbb{Z}$ und seine Nachbarknoten. Letztere Menge differenziert nicht zwischen Nachbarn durch eingehende und ausgehende Kanten, da diese Unterscheidung im Residualgraph entfällt.

### 3.1.2 Kanten

Die Klasse `class Edge` enthält alle Informationen über Kanten: Kosten, Kapazität, Fluss und die jeweiligen Endknoten. Residualkanten werden durch das Flag `isResidual`

gekennzeichnet. An dieser Stelle fällt auf, dass die Kapazität nicht nachträglich verändert werden kann, obwohl dies für Residualkanten notwendig wäre. Ich habe mich in meiner Implementierung dafür entschieden, auf Residualkanten Fluss zu erlauben und damit eine veränderliche Kapazität zu simulieren. Dies wird in Abschnitt 3.1.3 näher erläutert.

```
class Edge {
public:
  //edges are initialized with a flow of zero
  Edge (size_t node0, size_t node1, intmax_t cost,
        intmax_t capacity, bool isResidual = false);
  //returns true when flow change was successful
  bool changeFlow (intmax_t value);
  bool changeFlowPossible (intmax_t value);
  //toggles cost betweeen itself and 0
  void toggleCost ();

private:
  size_t node0, node1;
  intmax_t cost, capacity, flow, toggledCost;
  bool isResidual, isToggled = false;
};
```

Die Funktion `changeFlow` stellt über die Hilfsfunktion `changeFlowPossible` sicher, dass bei einer Flussveränderung keine Kapazitätsschranken verletzt werden. Mittels `toggleCost` können die Kosten einer Kante zwischen Null und ihrem eigentlichen Wert variiert werden. Somit lässt sich leicht die in Abschnitt 2.3.3 beschriebene Low-Cost-Initialisierung umsetzen, siehe dazu Abschnitt 3.3.3.

### 3.1.3 Graph

Werfen wir einen Blick darauf, wie die Knoten und Kanten in einem Netzwerk gespeichert werden. Ich habe mich dafür entschieden, die Objekte für eine logarithmische Laufzeit von `find`, `insert` sowie `delete` in einer `std::map` abzulegen. Die Knoten werden durch ihre `size_t id` identifiziert, Kanten wiederum durch Anfangsknoten, Endknoten und die `bool isResidual`. Eine `std::map` benötigt einen Vergleichsoperator, daher habe ich für die Kanten `custComp` definiert, der nach `size_t node0`, `size_t node1` und zuletzt `bool isResidual` vergleicht.

```
class Network {
private:
  intmax_t flow = 0, cost = 0;
  std::vector<size_t> sources, sinks, transit;
  std::map<std::tuple<size_t, size_t, bool>,
           Edge, custComp> edges;
  std::map<size_t, Node, std::less<size_t>> nodes;
};
```

Ein Netzwerk unterteilt zusätzlich seine Knoten nach ihrem b-Wert abzüglich des aktuellen Flusses in Quellen, Senken und Transitknoten. Eine Instanz mit einem maximalen Fluss besteht dementsprechend nur aus Transitknoten. Es kennt außerdem seinen Durchfluss `intmax_t flow`, also die Menge, die von den Quellen zu den Senken transportiert wird, und die dafür anfallenden Kosten `intmax_t cost`. Für den Algorithmus ist ersteres nicht relevant, da wir nur auf maximalen Flüssen arbeiten.

Der Graph $G$ mit einem Fluss $f$ und der dazugehörige Residualgraph $\bar{G}$ werden nicht als zwei verschiedene Objekte der Netzwerk-Klasse geführt, sondern in einer Instanz gemeinsam gespeichert. Um dies zu bewerkstelligen, wird bei jeder Veränderung an einer Kante gleichzeitig die eindeutige Residualkante aktualisiert. Zur Veranschaulichung folgt der Quellcode von `Network::addEdge` ohne die Prüfung der Zulässigkeit.

```
bool Network::addEdge(Edge e) {
  std::tuple<size_t, size_t, bool> key
    = std::make_tuple(e.node0, e.node1, e.isResidual);

  //insert both the edge and the residual edge
  edges.insert(std::make_pair(key, e));
  e.invert();
  edges.insert(std::make_pair(invertKey(key), e));

  nodes.find(node0)->second.neighbours.insert(node1);
  nodes.find(node1)->second.neighbours.insert(node0);
  return true;
}
```

Dabei vertauscht `invertKey` die beiden Knoten und negiert `isResidual`. Die oben nicht erwähnte Funktion `Edge::invert` spiegelt die Veränderungen von `invertKey` und negiert die Kosten gemäß Definition 8. Die Kapazität wird beibehalten, stattdessen wird der Fluss auf `flow = capacity - flow;` gesetzt. Der größte Wert, um den die Residualkante $\bar{e}$ augmentiert werden kann, ist also gleich $f(e)$. Sofern wir $f$ auf Nicht-Residualkanten einschränken, ist ein maximaler Fluss äquivalent zu Definition 2 implementiert. Dieser alternative Ansatz fordert noch ein wenig Aufmerksamkeit bei der Klassenfunktion `Network::changeFlow`,[1] danach können wir bei der Umsetzung des Algorithmus größtenteils ignorieren, ob eine Kante residual ist.

Das Netzwerk wird leer oder mit einer Anzahl `size_t noOfNodes` von Transitknoten initialisiert und durch Hinzufügen von Knoten und Kanten vervollständigt. Sofern keine Knoten gelöscht werden, ist die `size_t id` bei null beginnend lückenlos aufsteigend.

```
class Network {
public:
  Network(size_t noOfNodes);
```

[1] Für alle Kanten $e$ des Kreises wenden wir `e.changeFlow(f)` sowie `e.invert().changeFlow(-f)` an und aktualisieren die Kosten auf `this->cost += e.cost*f;`.

```
  bool addEdge(Edge e);
  //returns nodeID
  size_t addNode(intmax_t b_value = 0);

  //fails and returns 0 if there's flow left
  bool deleteEdge(size_t node0, size_t node1);
  //fails if there's flow on an edge to this node left
  bool deleteNode(size_t nodeID);

  //takes a path from a source to a sink
  //doesn't use residual edges
  bool addFlow(std::vector<size_t>& path, intmax_t flow);
  bool changeFlow(Circle& c, intmax_t flow);
};
```

Die Funktionen `deleteEdge` und `deleteNode` schlagen fehl, wenn auf den zu löschenden Kanten noch Fluss ist. So kann unter anderem am Ende des Algorithmus durch Löschen des künstlichen Knotens überprüft werden, ob die Instanz lösbar ist. Für Phase 1 des Algorithmus (siehe Abschnitt 3.3.3) nutzen wir die Funktion `addFlow`, die einen Weg von einer Quelle zu einer Senke verlangt.

In jeder Iteration der zweiten Phase des Netzwerk-Simplex-Algorithmus wird die Funktion `changeFlow` aufgerufen und augmentiert den übergebenen Kreis `Circle c` um einen beliebigen Wert, sofern dies möglich ist. Dieser Wert wird analog zu $\delta$ aus Kapitel 2 der größte zulässige sein. Die hier genutzte `class Circle` werden wir uns nun genauer anschauen.

## 3.2 Die Klasse *Circle*

Sei $G$ ein gerichteter Graph mit $n$ Knoten sowie $m$ Kanten und $T$ ein aufspannender Baum von $G$. In der Herleitung des Algorithmus haben wir genutzt, dass für eine Kante $e \in E(G) \backslash E(T)$ außerhalb des Baumes der Kreis $C_{T,e}$ eindeutig ist. Wird $T$ im Speicher gehalten, kann beispielsweise durch Tiefensuche in Laufzeit $\mathcal{O}(n)$ ein Kreis $C_{T,e}$ bzw. dessen Orientierung $\bar{C}_{T,e}$ gefunden werden.

Im Normalfall verändern sich von einer Iteration auf die nächste nur ein Bruchteil der Kreise. Diese lassen sich jedoch schlecht herausfiltern, wenn nur die Veränderung der zulässigen Baumlösung betrachtet wird. Um die Kreise nicht jede Iteration neu berechnen zu müssen, habe ich mich entschieden, nach [8] nicht den Baum abzuspeichern, sondern alle Kreise. Die theoretische Komplexität bleibt etwa dieselbe,[2] in der Praxis ist es schneller, nur die veränderten Kreise neu zu berechnen.

Die Klasse `class Circle` speichert gerichtete Kreise als zwei Vektoren, die Kanten wie gehabt über Anfangsknoten, Endknoten und die `bool isResidual` identifizieren. Außerdem speichern sie ihre jeweiligen Werte für $\delta$ in `flow` und $c(\bar{C}_{T,e})$ in `costPerFlow`;

[2] Wenn wir Pivotalgorithmen betrachten, die in Laufzeit $\mathcal{O}(\text{poly}(m-n))$ sämtliche Kreise in ihre Entscheidung einbeziehen, erhalten wir eine Komplexität von $\mathcal{O}(n \cdot (m-n) + \text{poly}(m-n))$.

diese müssen jedoch vom Algorithmus gesetzt werden. Nicht explizit in der Klassendeklaration schlägt sich die Invariante nieder, dass im Verlauf des Algorithmus jeder `Circle c` durch die erste Kante als einzige Nichtbaumkante spezifiziert wird.

```
//first edge of edges is not part of the underlying tree
class Circle {
public:
  //circles don't know about their graphs
  intmax_t flow = 0, costPerFlow = 0;

private:
  std::vector<std::pair<size_t, size_t>> edges;

  //std::vector<char>, because std::vector<bool> is broken[3]
  std::vector<char> isResidual;
};
```

Bevor wir uns die Umsetzung anschauen, überlegen wir uns theoretisch, was die Klasse leisten muss. Dazu betrachten wir eine Iteration, in der $C_1 = \bar{C}_{T,e_1}$ ausgewählt wurde. Sei $C_2 = \bar{C}_{T,e_2}$ mit $e_1 \neq e_2$ ein weiterer Kreis. Abb. 3.1 stellt für die zugrundeliegenden ungerichteten Kreise $\tilde{C}_1$ und $\tilde{C}_2$ dar, wie sich $C_1$ und $C_2$ zueinander verhalten können.

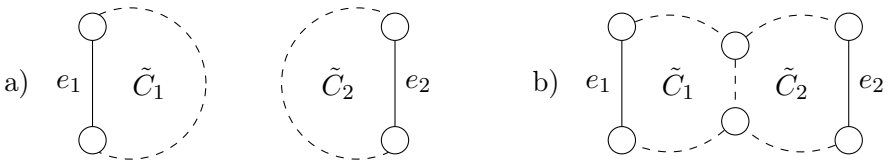


Abbildung 3.1: Beide Beispiele zeigen je zwei Kreise $\tilde{C}_i = C_{T,e_i}$. Die gestrichelten Linien gehören zum nicht vollständig dargestellten Baum $T$. Im zweiten Fall sind $\tilde{C}_1$ und $\tilde{C}_2$ nicht disjunkt.

Der Algorithmus wird nun $e_1$ zum neuen Baum $T'$ hinzufügen und eine ausgehende Kante $e_1 \neq e' \in E(C_{T,e_1})$ wählen, die dafür entfernt wird. Zunächst halten wir fest, dass es vor der Iteration keinen weiteren Fall als die beiden aus Abb. 3.1 geben kann: Wären $e_1$ und $e_2$ im Schnitt der ungerichteten Kreise $E(C_{T,e_1}) \cap E(C_{T,e_2})$ enthalten, dann wäre $T$ nicht kreisfrei.

Für den neuen ungerichteten Kreis gilt $C_{T',e'} = C_{T,e_1}$, zur Beibehaltung der Invariante muss `class Circle` die gespeicherten Kanten rotieren. Für die Konstellation, dass $e_1$ im Gegensatz zu $e'$ nicht saturiert ist oder vice versa ändert sich zusätzlich die Richtung von $\bar{C}_{T',e'}$ im Vergleich zu $\bar{C}_{T,e_1}$. All dies leistet die Funktion `Circle::rotateBy`:

```
void Circle::rotateBy (size_t index, bool toReverse) {
  //rotate, such that the correct edge is on front
```

[3] Für mehr Informationen siehe [9, *Item 18: Avoid using* `vector<bool>`].

```
  std::rotate(edges.begin(),
              edges.begin()+index, edges.end());
  std::rotate(isResidual.begin(),
              isResidual.begin()+index, isResidual.end());
  //all but the first entry might have to be reversed
  if (toReverse) {
    //change direction of all edges
    for (size_t i = 0; i < this->length; i++) {
      std::swap(edges[i].first, edges[i].second);
      isResidual[i] = not isResidual[i];
    }
    std::reverse(edges.begin() + 1, edges.end());
    std::reverse(isResidual.begin() + 1, isResidual.end());
    }
}
```

Der einfache Fall a) aus Abb. 3.1 ist damit abgehandelt, da $C_2$ unverändert bleibt. Dies gilt ebenfalls für Fall b), insofern $e' \notin E(C_{T,e_1}) \cap E(C_{T,e_2})$. Andernfalls wird, wie Abb. 3.2 veranschaulicht, $C_{T',e_2}$ nun $e_1$ oder $\varphi(e_1)$ statt $e'$ enthalten.

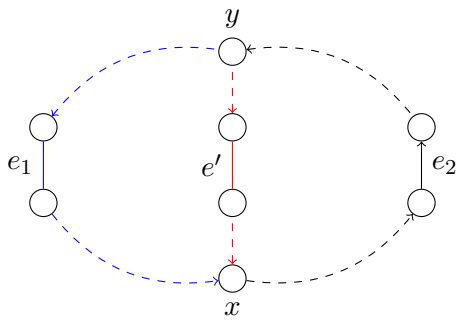


Abbildung 3.2: Ursprünglich sind $e_1$ und $e_2$ Nichtbaumkanten, die Orientierung von $e_1$ und $e'$ ist irrelevant. Der Kreis $\bar{C}_{T,e_2}$ geht von $x$ über $e_2$ nach $y$ (schwarze Kanten) zurück nach $x$ über $e'$ oder $\varphi(e')$ (rote Kanten). Im neuen Baum $T'$, der statt $e'$ die Kante $e_1$ enthält, führt $\bar{C}_{T',e_2}$ nun von $y$ nach $x$ über $e_1$ oder $\varphi(e_1)$ (blaue Kanten).

Sei $W := E(C_{T,e_1}) \cap E(C_{T,e_2})$ der Schnitt der beiden ungerichteten Kreise im alten Baum $T$. Dann gilt:

$$\begin{aligned} E(C_{T',e_2}) &= \big(E(C_{T,e_2}) \backslash W\big) \cup \big(E(C_{T,e_1}) \backslash W\big) \\ &= \big(E(C_{T,e_1}) \cup E(C_{T,e_2})\big) \backslash W \end{aligned} \tag{3.1}$$

Gemäß Gleichung (3.1) lässt sich also nach einer Iteration vom aufspannenden Baum $T$ zu $T' = T - \{e_1\} + \{e'\}$ die Kantenmenge jedes weiteren ungerichteten Kreises $C_{T,e_i}$ mit $i \neq 1$ aktualisieren, indem wir eine XOR-Operation mit der alten Kantenmenge

und $E(C_{T,e_1}) = E(C_{T',e'})$ durchführen. Um $\bar{C}_{T',e_i}$ zu erhalten, muss der Kreis nur noch entsprechend $e_i$ gerichtet werden.

Wir werden nun die recht komplexe Funktion `Circle::update`, die oben beschriebene Routine umsetzt, im Detail betrachten. Die Funktion bekommt `Circle& c` übergeben, dies entspricht $\bar{C}_{T',e'}$. Zunächst suchen wir sowohl in unseren Kanten als auch in ihren umgedrehten Pendants nach $e'$. Dies ist nötig, da wir die Orientierung der Kreise zueinander nicht kennen.

Sind wir nicht fündig geworden, ist die Routine abgeschlossen, da es nichts zu tun gibt. Ansonsten erreichen wir den Zustand, dass die beiden gerichteten Kreise gegeneinander orientiert sind, also der Schnitt ihrer Kanten leer ist, indem wir gegebenenfalls `c.rotateBy(0, true)` aufrufen. Damit ist gegeben, dass in `Circle c` nach Notation von Abb. 3.2 der Weg von $y$ nach $x$ den blauen Kanten entspricht.

```
void Circle::update(Circle& c) {
  bool iR = c.getIsResidual()[0];
  std::pair<size_t, size_t> edgeSame = c.getEdges()[0];
  std::pair<size_t, size_t> edgeReverse =
       std::make_pair(edgeSame.second, edgeSame.first);

  bool reversed = false;
  size_t index = 0;
  std::vector<std::pair<size_t, size_t>>::iterator
       it = edges.begin();
  //find edgeSame if existent
  for (index = 0; it != edges.end(); index++) {
    if (iR == isResidual[index] and
         edges[index] == edgeSame) {break;}
  it++;
  }
  //if edgeSame is in this->edges, reverse c,
  //do if-case and reverse back
  if (it != edges.end()) {
    c.rotateBy(0, true);
    reversed = true;
  }
  //else try to find edgeReverse
  else {
    it = edges.begin();
    for (index = 0; it != edges.end(); index++) {
      if (iR != isResidual[index] and
          edges[index] == edgeReverse) {break;}
      it++;
    }
  }
```

Insofern wir $e'$ oder $\varphi(e')$ gefunden haben, zeigt der Iterator `it` an dieser Stelle darauf. Indem wir von dieser Kante aus $x$ und $y$ suchen, finden wir heraus, welche Kanten in beiden zugrundeliegenden ungerichteten Kreisen enthalten sind. Insbesondere haben wir danach in `size_t indexLeft` und `indexRight` gespeichert, bis wohin bzw. ab wann wir die Kanten unseres bisherigen gerichteten Kreises beibehalten.

```
if (reversed or it != edges.end()) {
  std::vector<bool> toCopy (c.length, true);
  size_t indexLeft = index - 1, indexRight = index + 1;

  //take out all edges existent in both circles
  toCopy[0] = false; //first edge not even in tree
  //left from first edge
  for (; indexLeft > 0; indexLeft--) {
    //circles are in opposite directions,
    //so reverse the edge for existence check
    std::pair<size_t, size_t> checkEdge =
    std::make_pair(
         c.getEdges()[index - indexLeft].second,
         c.getEdges()[index - indexLeft].first);
    iR = c.getIsResidual()[index - indexLeft];
    if (iR != isResidual[indexLeft]
        and edges[indexLeft] == checkEdge)
    {toCopy[index - indexLeft] = false;}
    else {break;}
  }
  //right from first edge
  for (; indexRight < this->length; indexRight++) {
    //reverse the edge for existence check
    std::pair<size_t, size_t> checkEdge =
    std::make_pair(
         c.getEdges()[c.length+index-indexRight].second,
         c.getEdges()[c.length+index-indexRight].first);
    iR = c.getIsResidual()[c.length+index-indexRight];
    if (iR != isResidual[indexRight]
        and edges[indexRight] == checkEdge)
    {toCopy[c.length + index - indexRight] = false;}
    else {break;}
  }
```

Anschließend konstruiere ich der Einfachheit halber die beiden Vektoren `edges` und `isResidual` komplett neu. Der neue Kreis besteht dann wie in Abb. 3.2 aus dem Weg von $e_2$ nach $y$, gefolgt von den neuen, im Bild blauen Kanten von $y$ nach $x$ und dem Schluss von $x$ zu $e_2$. Falls wir `Circle c` umgedreht haben, machen wir dies als letztes noch rückgängig.

```
    //construct new circle
    std::vector<std::pair<size_t, size_t>> edgesNew;
    std::vector<char> isResidualNew;

    //first part of old circle
    for (size_t i = 0; i <= indexLeft; i++) {
      edgesNew.push_back(this->edges[i]);
      isResidualNew.push_back(this->isResidual[i]);
    }
    //bypass
    for (size_t i = 0; i < c.length; i++) {
      if (toCopy[i]) {
        edgesNew.push_back(c.getEdges()[i]);
        isResidualNew.push_back(c.getIsResidual()[i]);
      }
    }
    //last part of old circle
    for (size_t i = indexRight; i < this->length; i++) {
      edgesNew.push_back(this->edges[i]);
      isResidualNew.push_back(this->isResidual[i]);
    }

    this->edges = edgesNew;
    this->isResidual = isResidualNew;
  }

  //reverse back
  if(reversed) {c.rotateBy(0, true);}
}
```

Mit diesen Datenstrukturen als Grundlage können wir nun den Algorithmus implementieren.

## 3.3 Der Algorithmus

Die Klasse `class Algorithm` besteht nur aus einem Konstruktor und der Funktion `Algorithm::solution`, die das Min-Cost-Flow-Problem entweder mit LC- oder HC-Initialisierung (siehe Abschnitt 2.3.3 für die Funktionsweise und Abschnitt 3.3.3 für die Umsetzung) löst. Ein `Algorithm a` wird mit einer konkreten Pivotfunktion erstellt, die aus einem `std::vector<Circle>` einen Kreis auswählt und dessen Index zurückgibt.

```
class Algorithm {
public:
  Algorithm (Network& _n, size_t (*_pivot)
                          (const std::vector<Circle>&));
```

```
  //returns 0 if n cannot be solved
  //false for high cost edge initialization
  bool solution (bool modus = true);
}
```

Die Klasse speichert intern das Netzwerk, den Pivotalgorithmus, die Kreise, die `id` des künstlichen Knoten sowie die Anzahl an Iterationen ab. Zusätzlich gibt es eine Funktion zur anfänglichen Erzeugung aller Kreise per Tiefensuche.

```
class Algorithm {
private:
  intmax_t iterations = 0;
  Network& n;
  size_t (*pivot)(const std::vector<Circle>&);
  size_t artNode;
  std::vector<Circle> circles;

  //false if no optimization was possible
  bool optimize();
  //takes tree and creates circles;
  void createCircles(std::vector<Node> tree);
};
```

Der Kern der Klasse ist die Funktion `Algorithm::optimize`, die eine Iteration durchführt oder `false` zurückgibt, falls dies nicht mehr möglich und damit Phase 2 beendet ist. Nachdem Phase 1 abgeschlossen ist, wird dementsprechend die Schleife `while (optimize());` ausgeführt.

Die Funktion aktualisiert zunächst die Werte `flow` und `costPerFlow` für die Kreise. Dann wählt sie durch den Pivotalgorithmus einen Kreis `Circle c` für die nächste Iteration aus oder gibt `false` zurück, falls es keinen negativen Kreis mehr gibt. `c` wird um `c.flow` augmentiert und gemäß Abschnitt 2.3.1 wird die Kante $e'$ gewählt, die den Baum verlässt. Zum Schluss wenden wir `Circle::update` für alle anderen Kreise an. Mit unseren bisher vorgestellten Datenstrukturen und Codeschnipseln ist alles bis auf die Wahl von $e'$ leicht implementierbar. Das nächste Kapitel beschäftigt sich mit der Umsetzung dieses Teilproblems.

### 3.3.1 Stark zulässige Baumlösungen

Um die in Abschnitt 2.3.1 erklärte Regel der letzten blockierenden Kante umzusetzen, besitzt `class Algorithm` noch eine private Variable sowie zwei private Funktionen:

```
class Algorithm {
private:
  //for all nodes remember which edge leads to the root
  std::map<size_t, size_t> strongFeasibleTree;
```

```
  //functions used for last-blocking-edge-approach
  size_t findApex (Circle& c);
  void updateTree(Circle& c, size_t i, size_t apex);
};
```

Der Apex lässt sich schnell berechnen, wenn wir uns in einer `std::map` für jeden Knoten merken, welcher Nachbar auf dem Weg zum Wurzelknoten liegt. Am Anfang ist dieser Nachbar für jeden Knoten der künstliche Knoten, nach jeder Iteration wird `strongFeasibleTree` durch die Funktion `void updateTree` aktualisiert.

Beginnend mit dem zweiten Knoten eines `Circle c` genügt es nun, den Knoten zu finden, der in `strongFeasibleTree` einen anderen Nachbar hat als in `c`:

```
size_t Algorithm::findApex (Circle& c) {
  //find apex of circle
  size_t apex = 0;
  for (; apex < c.size() - 1; apex++) {
    size_t node = c.getEdges()[apex].second,
           neighbourInCircle = c.getEdges()[apex+1].second;
    //if the way to the root is leaving the circle
    if (strongFeasibleTree.find(node)->second !=
        neighbourInCircle) {break;}
  }
  return apex;
}
```

Sei `Circle chosenOne` der vom Pivotalgorithmus ausgewählte Kreis. Statt der letzten blockierenden Kante sucht meine Implementierung die erste in entgegengesetzter Richtung. Der folgende Code ist in `Algorithm::optimize` enthalten und terminiert notwendigerweise.

```
  size_t apex = findApex(chosenOne);
  for (size_t i = apex; ; i--) {
    const Edge& e = [4];

    if (e.flow == e.capacity) {
      updateStrongFeasibleTree(chosenOne, i, apex);

      //new first edge, but reversed direction
      chosenOne.rotateBy(i, true);
      break;
    }
  //go from first to last edge if necessary
  if (0 == i) {i = chosenOne.size();}
}
```

[4] `n.getEdges().find(std::forward_as_tuple(chosenOne.getEdges()[i].first, chosenOne.getEdges()[i].second, chosenOne.getIsResidual()[i]))->second`

### 3.3.2 Pivotalgorithmen

Die Pivotalgorithmen werden in meiner Implementierung in `PivotAlgorithms.h` gesammelt. Ein Pivotalgorithmus gibt einen ungültigen Index zurück, falls es keinen negativen Kreis gibt.

```
//return index of chosen circle or circles.size() if
//no circle can be chosen

//returns a random circle with negative cost
size_t pivotRandom(const std::vector<Circle>& circles);
//returns most negative circle
size_t pivotMaxVal(const std::vector<Circle>& circles);
//returns max |flow*costperflow|
size_t pivotMaxRev(const std::vector<Circle>& circles);
```

Beispielhaft schauen wir uns noch die Umsetzung von `pivotMaxVal` an:

```
size_t pivotMaxVal(const std::vector<Circle>& circles) {
  intmax_t mini = 0;
  size_t index = circles.size();
  for (size_t i = 0; i < circles.size(); i++) {
    intmax_t value = circles[i].costPerFlow;
    if (value < mini) {mini = value; index = i;}
  }
  return index;
}
```

### 3.3.3 Initialisierung

Wir haben nun alle essentiellen Bestandteil der Implementierung von Phase 2 des Netzwerk-Simplex-Algorithmus gesehen. Zum vollständigen Algorithmus fehlt uns nur noch Phase 1, wie in Abschnitt 2.3.3 beschrieben. Zu Beginn legen wir je nach Modus die Kosten der künstlichen Kanten fest.

```
bool Algorithm::solution (bool modus) {
  //high cost edge
  intmax_t maxCost = 0;
  if (modus == false) {
    for (auto[5] edge : n.getEdges()) {
      if (edge.second.cost > maxCost) {
        maxCost = edge.second.cost;
      }
    }
  }
```

[5] `const std::pair<const std::tuple<size_t, size_t, bool>, Edge>&`

```
//toggle all edges
else {n.toggleCost();}
maxCost = maxCost*n.getNoOfNodes() + 1;
```

Nun legen wir den künstlichen Knoten und die künstlichen Kanten an, hier beispielhaft für die Quellen:

```
artNode = n.addNode(0);

for (size_t source : n.sources) {
  n.addEdge(Edge(source, artNode, maxCost,
                 n.sumSource + 1));
}
```

Die Funktion `bool Algorithm::solution` gibt sofort `false` zurück, falls der übergebene Graph $G$ die Gleichung $\sum_{v \in V(G)} b(v) = 0$ nicht erfüllt. Somit können wir bei der Erzeugung des initialen Flusses $f_0$ Wege von Quellen zu Senken über den künstlichen Knoten augmentieren, bis der b-Wert eines der beiden Endknoten erfüllt ist, und dann mit der nächsten Quelle bzw. Senke iterieren.

```
//get that flow started
//networks updates n.sources after flow change
while (not n.sources.empty()) {
  size_t src = n.sources.back(), sink = n.sinks.back();
  if (std::abs(n.getNodes().find(src)->second.b_value)
   <= std::abs(n.getNodes().find(sink)->second.b_value)){
    std::vector<size_t> path = {src, artNode, sink};
    n.addFlow(path,
      std::abs(n.getNodes().find(src)->second.b_value));
  }
  else {
  std::vector<size_t> path = {src, artNode, sink};
  n.addFlow(path,
    std::abs(n.getNodes().find(sink)->second.b_value));
  }
}
```

Die Initialisierung ist an dieser Stelle abgeschlossen, nun können wir die Instanz lösen.

```
//complete tree and create circles
createCircles(tree);
//solve with artificial node
while (optimize());
```

Ist `modus = false`, dann befinden wir uns in HC und sind an dieser Stelle fertig. Kann der künstliche Knoten gelöscht werden, ist die Instanz lösbar und die Lösung im Netzwerk abgespeichert, andernfalls gibt `solution` den Wert `false` zurück.

Für LC wissen wir zunächst nur, ob die Instanz lösbar ist oder nicht. Wenn sie lösbar ist, können wir den künstlichen Knoten löschen, über `Network::toggleCost` die originalen Kosten wiederherstellen und mit `optimize` eine optimale Lösung finden.

Um die dafür notwendige stark zulässige Baumlösung auf dem Netzwerk ohne künstlichen Knoten zu finden, gibt es zwei Ansätze: Ein Weg ist es, einfach auf dem Graphen einen neuen Baum $T'$ zu bestimmen, der mit dem aktuellen Fluss $f$ eine stark zulässige Baumlösung $(T', f)_{r'}$ bildet. In meiner Umsetzung führen wir dagegen vor der Löschung des künstlichen Knoten degenerierte Iterationen durch, bis dieser ein Blatt ist. Dies schien mir der kohärente Ansatz zu sein.

Dafür gehen wir über den Vektor `circles` und suchen nach Kreisen, in denen zwei Baumkanten mit dem künstlichen Knoten benachbart sind. Sei $e$ diejenige der beiden Kanten, die den Baum verlässt. $e$ kann nur in schon betrachteten Kreisen enthalten sein, wenn deren identifizierende Kante mit dem künstlichen Knoten benachbart ist; diese Kreise werden jedoch später gelöscht. Also genügt es, nur Kreise über `update` zu aktualisieren, die noch nicht betrachtet wurden, womit folgende Subroutine terminiert.

```
for (std::vector<Circle>::iterator it = circles.begin();
     it != circles.end(); it++) {
  //find edge over artificial node if existent
  size_t i = 1;
  for (; i < it->size(); i++) {
    if (it->getEdges()[i].first == artificialNode or
        it->getEdges()[i].second == artificialNode)
       {break;}
  }
  if (i < it->size()) {
    updateStrongFeasibleTree(*it, i, findApex(*it));
    //since n is feasible, one of the both cases occures

    //not residual <==> flow == 0
    //new first edge, same direction
    if (not it->getIsResidual()[i])
       {it->rotateBy(i, false);}
    //new first edge, but reversed direction
    else {it->rotateBy(i, true);}

    //circles before this one don't need an update
    for (std::vector<Circle>::iterator otherCircle =it+1;
         otherCircle != circles.end(); otherCircle++)
        {otherCircle->update(*it);}
    iterations++;
  }
```

Zuletzt werden der künstliche Knoten und wie angedeutet die $|V(G)| - 1$ Kreise gelöscht, die am künstlichen Knoten beginnen, bevor die finale Lösung berechnet

wird. Ich benutze an dieser Stelle eine Lambdafunktion, um die entsprechenden Kreise herauszufiltern.

```
  //now remove all circles beginning at the artificial node
  circles.erase(std::remove_if(circles.begin(),
                               circles.end(),
    [this](const Circle& c)
    {return c.getEdges()[0].first == artificialNode or
            c.getEdges()[0].second == artificialNode;}
    ), circles.end());

  while (optimize());
  return true;
}
```

# 4 Exponentiellen Instanzen nach Zadeh

Zadeh veröffentlichte 1973 in [10] eine exponentielle Instanzenreihe des Transportproblems für diverse verbreitete Algorithmen. Dabei wählte er im Falle des Netzwerk-Simplex-Algorithmus eine initiale stark zulässige Baumlösung, die an den Baum der High-Cost-Initialisierung erinnert. Damit können wir rekursiv Netzwerke $N_j$ für $j \geq 3$ mit $2j$ Knoten definieren, für die der programmierte Algorithmus mit HC und MaxVal $2^j + 2^{j-1} - 2$ Iterationen benötigt. Abb. 4.1 zeigt den Rekursionanfang $N_3$.

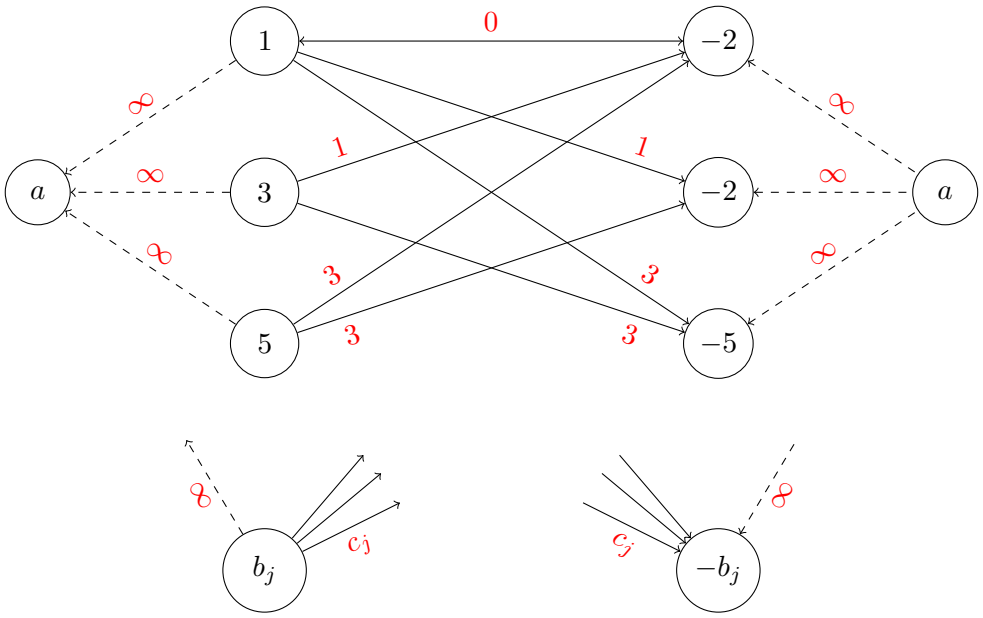


Abbildung 4.1: Grafik angelehnt an [10, S. 261], alle Kapazitäten sind unendlich. Die sechs mittleren Knoten bilden das Netzwerk $N_3$, die gestrichelten Linien stellen den initialen Baum dar, wobei die beiden Knoten $a$ identisch sein sollen. Die rekursive Erzeugung des Netzwerkes $N_j$ wird angedeutet.

Es folgt zunächst eine beispielhafte Implementierung des Netzwerkes $N_3$, bevor wir die Rekursion betrachten und umsetzen:

```
Network n3 = Network(0);
std::vector<intmax_t> b_value = {1, -2, 3, -2, 5, -5};
for (intmax_t b : b_value) {n3.addNode(b);}

intmax_t u = 1000; //"infinite" capacity
```

```
std::vector<size_t> from = {0,1,0,0,2,2,4,4};
std::vector<size_t> to   = {1,0,3,5,1,5,1,3};
std::vector<intmax_t> co = {0,0,1,3,1,3,3,3};
std::vector<intmax_t> ca = {u,u,u,u,u,u,u,u};

for (size_t i = 0; i < from.size(); i++) {
  n3.addEdge(Edge(from[i], to[i], co[i], ca[i]))
}
```

Wie Abb. 4.1 andeutet, entsteht das Netzwerk $N_j$, indem wir zu $N_{j-1}$ zwei neue Knoten $q_j$ und $s_j$ mit b-Wert $b_j := 2^{j-1} + 2^{j-3}$ und $-b_j$ hinzufügen. Wir fixieren $c_j := 2^{j-1} - 1$ als Kosten. Die Quelle $q_j$ verbinden wir mit jeder Senke $s_i \neq s_j$ über eine Kante mit Kosten $c((q_j, s_i)) = c_j$, die Senke ist von jeder Quelle $q_i \neq q_j$ aus mit Kosten $c((q_i, s_j)) = c_j$ erreichbar.

Wenn wir bereits ein `Network n` haben, das $N_{j-1}$ entspricht, führt uns folgender Code zu $N_j$. Es wird davon ausgegangen, dass für eine Quelle $q_i$ `id = 2*i` und für eine Senke $s_j$ `id = 2*i + 1` gilt. Außerdem sollte `intmax_t u` hinreichend groß sein.

```
size_t j = n.getNoOfNodes()/2 + 1;
intmax_t b_value = std::pow(2, j-1) + std::pow(2, j-3);
size_t q_j = n.addNode(b_value), s_j = n.addNode(-b_value);

intmax_t c_j = std::pow(2, j-1) - 1;
for (size_t i = 0; i < j-1; i++) {
  test2.addEdge(Edge(q_j, 2*i + 1, c_j, u));
  test2.addEdge(Edge(2*i, s_j, c_j, u));
}
```

# 5 Experimentelle Ergebnisse

Meh.

# 6 Ausblick

La la la.

# Literaturverzeichnis


[1] V. Chvátal, *Linear Programming*, pp. 291 ff. Series of books in the mathematical sciences, W. H. Freeman, 16 ed., 2002.

[2] G. B. Dantzig, "Application of the simplex method to a transportation problem," in *Activity Analysis of Production and Allocation* (T. C. Koopmans, ed.), ch. XXIII, pp. 359–373, New York: Wiley, 1951.

[3] A. Orden, "The transhipment problem," *Management Science*, vol. 2, no. 3, pp. 276–285, 1956.

[4] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*, pp. 453 ff. Hoboken, New Jersey: John Wiley & Sons, Inc., 4 ed., 2010.

[5] S. Hougardy and J. Vygen, *Algorithmische Mathematik*. Springer-Lehrbuch, Berlin/Heidelberg: Springer Spektrum, 2015.

[6] W. H. Cunningham, "A network simplex method," *Mathematical Programming*, vol. 11, no. 1, pp. 105–116, 1976.

[7] D. Jungnickel, *Graphs, Networks and Algorithms*. No. 5 in Algorithms and Computation in Mathematics, Heidelberg u. a.: Springer, 4 ed., 2013.

[8] X. Zhong. Persönliche Mitteilung, 2018.

[9] S. Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 1 ed., 2001.

[10] N. Zadeh, "A bad network problem for the simplex method and other minimum cost flow algorithms," *Mathematical Programming*, vol. 5, no. 1, pp. 255–266, 1973.