

Eine graphentheoretische Herleitung und Implementierung des Netzwerk-Simplex-Algorithmus

Max Kanold

Geboren am 5. Juni 1995 in Dresden

26. November 2018

Bachelorarbeit Informatik

Betreuer: Prof. Dr. Heiko Röglin

Zweitgutachter: Priv.-Doz. Dr. Elmar Langetepe

INSTITUT FÜR INFORMATIK

MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT DER
RHEINISCHEN FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

An dieser Stelle möchte ich mich herzlich bei Xianghui Zhong und Heiko Röglin für die hilfreichen Anregungen und die unkomplizierte Betreuung der Bachelorarbeit bedanken. Ein besonderer Dank gebührt den unermüdlichen Korrekturlesern Debbora Leip und Jonathan Gruner, sämtliche verbliebenen Fehler gehen nun auf meine Kappe. Zu guter Letzt danke ich dem zweiten und dritten Mann meiner Skatrunde dafür, dass sie zu jeder Schandtat bereit waren.

Inhaltsverzeichnis

1	Einführung	1
2	Der Netzwerk-Simplex-Algorithmus	2
2.1	Min-Cost-Flow-Problem	2
2.2	Zulässige Baumlösungen	3
2.3	Der grundlegende Algorithmus	7
2.3.1	Degenerierte Iterationen	12
2.3.2	Pivotalgorithmen	14
2.3.3	Initialisierung	15
2.4	Erweiterung auf beschränkte Kapazitäten	16
3	Implementierung	19
3.1	Netzwerke	19
3.1.1	Knoten	19
3.1.2	Kanten	19
3.1.3	Graph	20
3.2	Die Klasse <i>Circle</i>	22
3.3	Der Algorithmus	27
3.3.1	Stark zulässige Baumlösungen	28
3.3.2	Pivotalgorithmen	30
3.3.3	Initialisierung	30
4	Exponentielle Instanzen nach Zadeh	34
5	Experimentelle Ergebnisse	37
6	Ausblick	39
	Literaturverzeichnis	40

1 Einführung

Sehr viele Probleme in der Informatik lassen sich als Lineares Programm darstellen, so auch das in Abschnitt 2.1 eingeführte Min-Cost-Flow-Problem aus dem Bereich der Graphentheorie. Logistische und kombinatorische Fragestellungen der Praxis lassen sich wiederum durch dieses modellieren. Das Simplex-Verfahren, zu welchem eine Einführung in [3] gefunden werden kann, löst Lineare Programme in der Anwendung sehr schnell, obwohl die Worst-Case-Laufzeit nicht polynomiell ist.

Dantzig und Orden vereinfachten in den 1950er Jahren mit [5] bzw. [9] das Simplex-Verfahren für die konkrete Struktur des Min-Cost-Flow-Problems zum Netzwerk-Simplex-Algorithmus. Gemäß [1, S. 445] ist diese Vereinfachung um einen Faktor von 200–300 schneller, außerdem lässt sich der Netzwerk-Simplex-Algorithmus rein graphentheoretisch definieren.

In dieser Bachelorarbeit wird in Kapitel 2 die graphentheoretische Herleitung mit Beweis der Korrektheit vollzogen und anschließend in Kapitel 3 die Implementierung des Algorithmus in C++ dargelegt. Insgesamt wurden sechs Varianten umgesetzt. Für eine von diesen veröffentlichte Zadeh 1973 in [10] eine Familie von exponentiellen Instanzen; diese wird in Kapitel 4 vorgestellt und verifiziert. Kapitel 5 greift die Frage auf, ob sich allgemein mit einfachen algorithmischen Ansätzen schlechte Instanzen für beliebige Versionen des Netzwerk-Simplex-Algorithmus finden lassen. Hierzu muss an dieser Stelle ein negatives Ergebnis verzeichnet werden. Kapitel 6 skizziert als Abschluss Vorschläge für eine Vertiefung der Gesamtthematik.

Im weiteren Verlauf dieser Bachelorarbeit ist mit Laufzeit des Algorithmus nicht die tatsächlich benötigte Zeit, sondern die in Abschnitt 2.3 eingeführte Anzahl an internen Iterationen gemeint. Sämtliche Abbildungen von Graphen im Verlauf dieser Bachelorarbeit sind entsprechend der folgenden Legende zu lesen. Teilweise sind die Graphen der Übersichtlichkeit halber unvollständig beschriftet, die Kapazität tritt jedoch nie ohne Flusswerte auf.

- 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

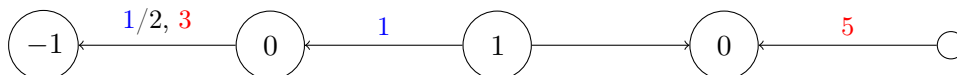


Abbildung 1.1: Beispielnetzwerk mit verschiedenen Spezifikationen.

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 [3, S. 291 ff.] für die Beschreibung des Netzwerk-Simplex-Algorithmus zur Lösung des Transportproblems, danach wird in Abschnitt 2.4 der Algorithmus anhand von [3, 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) aus einem gerichteten Graphen $G = (V, E)$ und den Abbildungen $b: V \rightarrow \mathbb{R}$, $c: E \rightarrow \mathbb{R}$ sowie $u: E \rightarrow \mathbb{R}_{\geq 0} \cup \{\infty\}$. 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* und die mit neutralem b-Wert als *Transitknoten* bezeichnet.

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

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

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 dominieren, ist die Instanz unlösbar. Der umgekehrte Fall kann durch eine Dummy-Senke¹ 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 [6] 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.

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 $e \in E(G) \setminus E(T)$ außerhalb des Baumes $f(e) = 0$ gilt.

¹Gemäß [1, 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 $u(e_i) = \infty$.

2 Der Netzwerk-Simplex-Algorithmus

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 eindeutige 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 verändert werden:

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

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

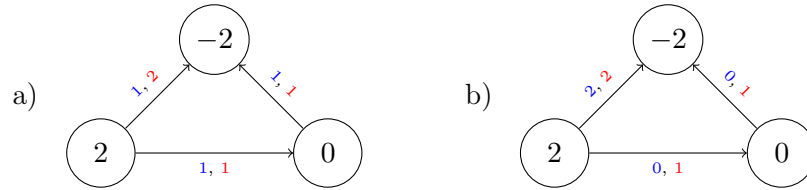


Abbildung 2.1: Links ein maximaler Fluss ohne zulässige Baumlösung, rechts ein maximaler Fluss mit mehreren zulässigen Baumlösungen.

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 sein muss.

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* $u_f: \bar{E} \rightarrow \mathbb{N}_{\geq 0}$ ist bestimmt durch $u_f(\bar{e}) = f(e)$, die *Residualkosten* $c_f: \bar{E} \rightarrow \mathbb{Z}$ durch $c_f(\bar{e}) = -c(e)$.

Das **Residualnetzwerk** R ist ein Tupel $R_{N,f} = (\bar{G} = (V, E \amalg \bar{E}), \bar{f}, b, \bar{c}, \bar{u})$ aus 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 $\bar{c} = c \amalg c_f: E \amalg \bar{E} \rightarrow \mathbb{Z}$ und der Kapazitätsfunktion $\bar{u} = u \amalg u_f: E \amalg \bar{E} \rightarrow \mathbb{N}_{\geq 0} \cup \{\infty\}$.

Notation. Mit $\varphi: E \leftrightarrow \bar{E}$ 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 δ 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:

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

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, C ein gerichteter Kreis mit negativen Kosten im Residualgraph \bar{G} und δ der größte Wert, um den C augmentiert werden kann. Dann ist δ endlich und nach der Augmentierung um δ 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.

2 Der Netzwerk-Simplex-Algorithmus

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 $c: E(G) \rightarrow \mathbb{N}_{\geq 0}$ 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 δ 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 δ zu einem maximalen Fluss f' augmentiert haben, gilt für die korrespondierende Kante $e := \varphi(\bar{e})$:

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

Notation. Sei f ein maximaler Fluss auf einem 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 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 $c(\hat{f}) \leq c(f)$ gilt und eine zulässige Baumlösung (\hat{T}, \hat{f}) gefunden werden kann.*

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 $f = f_0, f_1, f_2, \dots, f_n = \hat{f}$ 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 $C \not\subseteq G_{f_{i+1}}$ 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 $C \not\subseteq G_{f_{i+1}}$ 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 $\hat{f} := f_j$. \square

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

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 $e \in E(\bar{G}) \setminus E(T)$ 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 $e \in E(G) \setminus E(T)$ 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, T ein Baum in 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. 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) $\forall e \in E(C) \setminus E(T)$: die zu e korrespondierende Kante $\bar{e} \in E(\bar{C})$ darf vom Netzwerk-Simplex-Algorithmus bei zulässiger Baumlösung (T, f) als eingehende Kante in Betracht gezogen werden

2 Der Netzwerk-Simplex-Algorithmus

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 $E_{\bar{C},-T} \subseteq E(\bar{C})$ die Menge der zu den Nichtbaumkanten $E(C) \setminus E(T)$ korrespondierenden Kanten in \bar{C} und $E_{\bar{C},T} = E(\bar{C}) \setminus E_{\bar{C},-T}$ dessen Komplement. Betrachten wir zunächst den Fall, dass $E_{\bar{C},T} = \emptyset$:

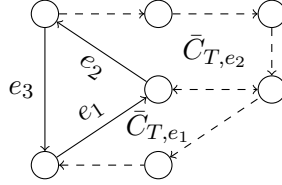


Abbildung 2.2: Die durchgezogenen Kanten bilden den negativen Kreis \bar{C} . Jede Kante $(v, w) \in E(\bar{C})$ wird über $w \xrightarrow{T} v$ zu einem Kreis ergänzt.

Nach Eigenschaft (ii) gibt es für jede Kante $e_i \in E_{\bar{C},-T} = E(\bar{C})$ 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 dem Kreis \bar{C} anfangen, 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:

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

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

Betrachte einen Iterationsschritt mit negativem Kreis \bar{C} , sodass $|E_{\bar{C},-T}| > 1$ gilt. Sei $e = (x, y) \in E_{\bar{C},-T}$ 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, \dots, C_w mit $c(W) = \sum_{i=1}^w c(C_i)$. Der Knoten x sei o. B. d. A. in C_1 enthalten.

Sollten die Kosten $c(\tilde{C}) < 0$ negativ sein, ist \tilde{C} nach Eigenschaft (ii) der gesuchte Kreis. Ist einer der Kreise C_2, \dots, C_w negativ, ersetzen wir \bar{C} durch diesen Kreis.

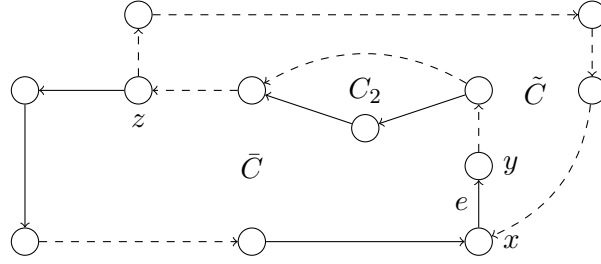


Abbildung 2.3: Gestrichelte Linien sind zum unvollständig dargestellten Baum T assoziiert. Der neue Kreis C_1 entsteht, indem wir von x beginnend entgegen \tilde{C} bis z gehen und dann dem bisherigen Kreisverlauf \bar{C} folgen.

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. \square

Um wie gewünscht 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 $z: E(\bar{G}) \rightarrow \mathbb{N}_{\geq 0}$, die folgende Eigenschaften erfüllt:

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

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

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 $\mathcal{Z} = \{z_1, \dots, z_n\}$ von Zirkulationen, sodass für alle $z_i \in \mathcal{Z}$ 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.

2 Der Netzwerk-Simplex-Algorithmus

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 durchflossener 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 $|E(\bar{G}_{z'})| \leq |E(\bar{G}_z) \setminus \{\tilde{e}\}| < |E(\bar{G}_z)|$. Wir nehmen per Induktion an, dass wir für z' eine Menge $\mathcal{Z}' := \{z'_1, \dots, z'_n\}$ 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 $\mathcal{Z} := \mathcal{Z}' \cup \{z_{n+1}\}$, womit für z und \mathcal{Z} nach Konstruktion gilt:

$$\begin{aligned} \forall e \in E(\bar{G}) \setminus E(C): \quad \sum_{z_i \in \mathcal{Z}} z_i(e) &= \sum_{z'_i \in \mathcal{Z}'} z'_i(e) + z_{n+1}(e) = z'(e) + 0 = z(e) \\ \forall e \in E(C): \quad \sum_{z_i \in \mathcal{Z}} z_i(e) &= \sum_{z'_i \in \mathcal{Z}'} z'_i(e) + z_{n+1}(e) = z'(e) + \delta = z(e) \end{aligned}$$

Für die Kosten ergibt sich:

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

Damit ist die Aussage für alle Zirkulationen gezeigt. \square

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 $z: E(\bar{G}) \rightarrow \mathbb{N}_{\geq 0}$ 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:

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

Die erste Gleichheit folgt dabei aus der 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. Abschließend betrachten wir die Kosten von z :

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

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. \square

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.*

2 Der Netzwerk-Simplex-Algorithmus

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 $\mathcal{Z} = \{z_1, \dots, z_n\}$ von Zirkulationen, sodass $c(z) = \sum_{i=1}^n c(z_i)$ gilt und \bar{G}_{z_i} für alle $z_i \in \mathcal{Z}$ ein gerichteter Kreis ist.

Da $c(z) < 0$ ist, gibt es mindestens eine Zirkulation $z_j \in \mathcal{Z}$ mit negativen Kosten. Sei $C_j := \bar{G}_{z_j}$ der von ihr durchflossene Kreis und $e \in E(C_j)$ eine der Kanten. 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. \square

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

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 dies nicht der Fall ist, sondern nur der Baum verändert wird. 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**.

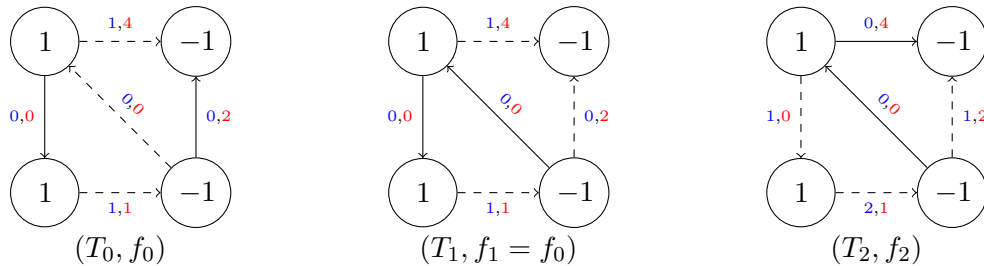


Abbildung 2.4: Die Kanten des Baumes T_i sind gestrichelt dargestellt, der gewählte negative Kreis ist jeweils eindeutig. Die Startlösung kann erst nach einer degenerierten Iteration verbessert werden.

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 *degenerierte Baumlösung* 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 [3, S. 303].

In [4] 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, T ein aufspannender Baum von G , f ein maximaler Fluss auf N und $r \in V(T)$ ein beliebiger Knoten. Dann bezeichnen wir $(T, f)_r$ als **stark zulässige Baumlösung**, falls (T, f) eine zulässige Baumlösung ist und zusätzlich jede Kante $e = (v, w) \in E(T)$ mit $f(e) = 0$ von r wegführt, also e im Weg $r \xrightarrow{T} w$ enthalten ist. r wird in dem Fall *Wurzel* genannt.

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

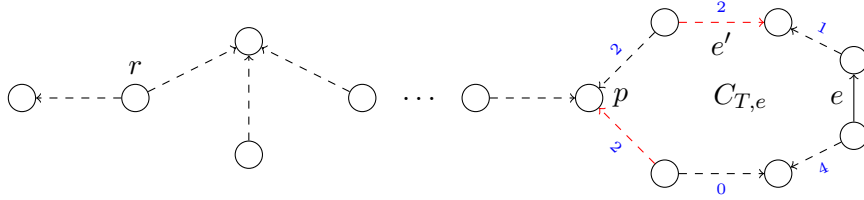


Abbildung 2.5: Grafik angelehnt an [7, S. 372]. Die gestrichelten Linien stellen den Baum T mit Wurzel r dar. Der Apex von $C_{T,e}$ ist p .

In einem Kreis $C_{T,e}$ bezeichnen wir als *blockierende Kanten* solche, die verhindern, dass der Kreis um mehr als δ augmentiert werden kann. Der Algorithmus wählt als zu entfernende Kante immer eine von diesen. In Abb. 2.5 ist $\delta = 2$ und die blockierenden Kanten sind rot eingefärbt.

Die von Cunningham eingeführte *Regel der letzten blockierenden Kante* besagt 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 und wird im Normalfall sogar beschleunigt.

Lemma 21 [4, 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, so ist die entstehende Baumlösung (T', f') wieder eine stark zulässige Baumlösung $(T', f')_r$.

Beweis. Für den Beweis siehe [4, S. 108 f.]. □

Theorem 22 [4, S. 108 f.]. *Der Netzwerk-Simplex-Algorithmus mit einer initialen stark zulässigen Baumlösung und der Regel der letzten blockierenden Kante terminiert.*

Beweis. Für den Beweis siehe [4, S. 108 f.]. \square

Nach [7, S. 359] ist die Laufzeit des Netzwerk-Simplex-Algorithmus für bestimmte Varianten polynomiell in der Anzahl an Knoten und Kanten sowie den höchsten Kosten einer Kante beschränkt. Für die in der Praxis verwendeten Varianten sind keine polynomiellen Schranken bekannt, teilweise wurden sogar exponentielle Instanzen gefunden. Diese basieren auf einer exponentiellen Anzahl degenerierter Iterationen, *Stalling* genannt. Mit einer solchen Familie von Instanzen werden wir uns in Kapitel 4 näher beschäftigen. Kapitel 5 befasst sich experimentell mit der algorithmischen Suche nach schlechten Instanzen.

Zunächst vervollständigen wir jedoch den Algorithmus. Dazu diskutieren wir 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 möglichen 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, den negativsten Kreis zu wählen, das heißt:

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

Wir werden diesen Weg mit *MaxVal* bezeichnen.

Maximum Revenue

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

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

Nach Lemma 9 ist jedes δ_e endlich. Sollten nur degenerierte Iterationen zur Auswahl stehen, ist $c(\bar{C}_{T,e}) = 0$ für alle $\bar{C}_{T,e} \in \bar{C}_T$. In dem Fall wendet meine Implementierung *MaxVal* an; hier sind aber auch andere Strategien denkbar.

Random

In einem überraschend effektiven Ansatz, mit *Random* bezeichnet, wird $\tilde{C} \in \bar{C}_T$ zufällig gewählt. Für diese Variante ist es schwierig, die erwartete Laufzeit zu bestimmen.

2.3.3 Initialisierung

Für einen vollständigen Algorithmus zur Lösung des Transportproblems fehlt nur noch die initiale stark zulässige Baumlösung $(T_0, f_0)_r$. Um diese zu finden, 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 Knoten bezeichnet wird. Für alle Quellen $v \in V(G)$ mit $b(v) > 0$ ergänzen wir eine künstliche Kante (v, a) , für alle Senken und Transitknoten $a \neq v \in V(G)$ mit $b(v) \leq 0$ eine künstliche Kante (a, v) . Sowohl die Kapazität als auch die Kosten der künstlichen Kanten e_v sind unendlich, letztere können in der Praxis 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. Sei G' der entstehende Graph.

Ohne weitere Probleme lässt sich nun die stark zulässige Baumlösung $(T_0, f_0)_a$ finden: Wir setzen $T_0 := (V(G), \{e_v \mid v \in V(G)\})$, dann ist nach Lemma 7 f_0 eindeutig. Diese Art der Initialisierung wird 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.

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

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

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

Offensichtlich ist N genau dann lösbar, wenn \tilde{N} lösbar ist. Wenn $c(f') > 0$ ist, enthält $G'_{f'}$ eine künstliche Kante. Damit ist \tilde{N} nach Lemma 23 nicht lösbar und der Algorithmus wird beendet. Andernfalls iteriert er $(T', f')_a$ degeneriert, bis a ein Blatt von T' ist. Dies wird in Abschnitt 3.3.3 detailliert erläutert.

Dadurch erhalten wir die zulässige Baumlösung $(T = T' - \{a\}, f = f'|_{E(G)})$ für \tilde{N} und damit für N . Als neue Wurzel können wir den eindeutigen Nachbar k_a von a in

T' wählen, was uns zur stark zulässigen Baumlösung $(T, f)_{k_a}$ führt. Nun kann eine optimale Lösung des Transportproblems für N berechnet werden.

2.4 Erweiterung auf beschränkte Kapazitäten

Bisher war nur die Kapazität von Residualkanten beschränkt. Wir werden nun den Netzwerk-Simplex-Algorithmus vom Transportproblem auf das allgemeinere Min-Cost-Flow-Problem erweitern. Dazu verallgemeinern wir die bisher eingeführten Definitionen, die resultierende Funktionsweise bleibt dieselbe.

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) , falls für alle Kanten $e \in E(G) \setminus E(T)$ 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, T ein aufspannender Baum von G , f ein maximaler Fluss auf N und $r \in V(T)$ eine Wurzel. Dann bezeichnen wir $(T, f)_r$ als **stark zulässige Baumlösung**, falls (T, f) eine zulässige Baumlösung ist und zusätzliche für jede Kante $e = (v, w) \in E(T)$ gilt:

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

Anmerkung. Kanten $e \in E(G)$ mit $u(e) = 0$ müssen an dieser Stelle aus dem Graphen entfernt werden, da sie im Widerspruch zur Definition stehen. Dies stellt keine Einschränkung dar, denn für das Problem sind sie ohnehin irrelevant.

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 $e \in E(G) \setminus E(T)$ 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 dieser negativ ist. Für stark zulässige Baumlösungen müssen zusätzlich alle saturierten Kanten des Baumes zur Wurzel hinführen.



Abbildung 2.6: Zwei unterschiedliche maximale Flüsse auf demselben, durch gestrichelte Linien dargestellten Baum. Die Senke entspricht der Wurzel.

Abb. 2.6 illustriert des Weiteren, 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 dennoch identisch durchführbar ist.

Die in Abschnitt 2.3.1 beschriebene Regel der letzten blockierenden Kante wurde in [4] ohnehin für den allgemeinen Fall eingeführt und lässt sich einfach übertragen. Wir werden im Folgenden Lemma 9, Theorem 10, Korollar 11 und Theorem 12 auf das Min-Cost-Flow-Problem erweitern.

Lemma 26. *Sei N ein Netzwerk, $R_{N,f}$ sein Residualnetzwerk, C ein gerichteter Kreis mit negativen Kosten im Residualgraph \bar{G} und δ der größte Wert, um den C augmentiert werden kann. Dann ist δ endlich und nach der Augmentierung um δ 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 $\bar{e} := \arg \min_{e \in E(C)} \{\bar{u}(e) - \bar{f}(e)\}$ und $\delta := \bar{u}(\bar{e}) - \bar{f}(\bar{e})$. Analog zum Beweis von Lemma 9 ergibt sich, dass δ endlich ist.

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

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

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

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

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 $H^f = H_f \setminus \{e \in E' \mid f(e) = u(e)\}$ 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 $c(\hat{f}) \leq c(f)$ gilt und eine zulässige Baumlösung (\hat{T}, \hat{f}) gefunden werden kann.*

Beweis. Sei $N = (G, b, c, u)$ ein Netzwerk mit maximalen Fluss f . Diesmal wird f so zu einem maximalen Fluss \hat{f} umgewandelt, dass $G^{\hat{f}}$ ein Wald ist. Der Beweis verläuft nun analog zum Beweis von Theorem 10, statt Lemma 9 wenden wir Lemma 26 an. \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 analog zum Beweis von Theorem 12 verfahren und zeigen, dass ein negativer Kreis $\bar{C}_{T,e}$ existiert, falls 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. Wir betrachten die

2 Der Netzwerk-Simplex-Algorithmus

Zirkulation $z := \hat{f} - f$ und ihre Zerlegung gemäß Lemma 15 in eine Menge \mathcal{Z} von Zirkulationen über gerichtete Kreise. Erneut gibt es eine Zirkulation $z_j \in \mathcal{Z}$ mit negativen Kosten, sei $C_j = \bar{G}_{z_j}$ der ihr zugehörige negative Kreis.

Sei weiterhin $e \in E(C_j)$ 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 hiermit gegeben. \square

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

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 beigelegte CD konsultiert werden. Im Folgenden wurden an einigen Stellen für ein besseres Layout Variablenamen 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`

3 Implementierung

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 zuzulassen 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 between 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

Betrachten wir nun, 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. Dazu 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 in der Implementierung äquivalent zu Definition 2. 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);
```

¹Für 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`;

3 Implementierung

```
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 löschen den 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 Netzwerk-Simplex-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 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 δ 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 $e \in E(G) \setminus E(T)$ außerhalb des Baumes der Kreis $C_{T,e}$ eindeutig ist. Wird T im Speicher gehalten, kann beispielsweise durch Tiefensuche in Laufzeit $\mathcal{O}(n)$ der 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 [11] 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, welche Kanten wie gehabt über Anfangsknoten, Endknoten und die `bool isResidual` identifizieren. Außerdem speichern sie ihre jeweiligen Werte für δ in `flow` und $c(\bar{C}_{T,e})$ in `costPerFlow`;

²Wenn 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 broken3
    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 $e_1 \neq e_2$ und $C_2 = \bar{C}_{T,e_2}$ 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.

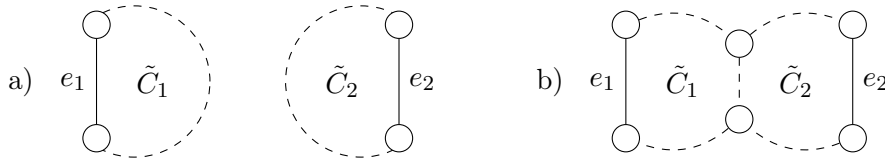


Abbildung 3.1: Beide Beispiele zeigen je zwei Kreise $\tilde{C}_i = C_{T,e_i}$. Die gestrichelten Linien gehören zum nicht vollständig dargestellten Baum T . Im zweiten Fall sind \tilde{C}_1 und \tilde{C}_2 nicht disjunkt.

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, so 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 den Fall, dass e' entgegen dem Kreis \bar{C}_{T',e_1} orientiert ist, 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
```

³Für mehr Informationen siehe [8, Item 18: Avoid using `vector<bool>`].

3 Implementierung

```

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 einfachere 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.

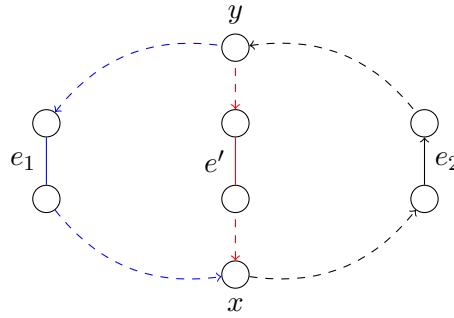


Abbildung 3.2: Ursprünglich sind e_1 und e_2 Nichtbaumkanten, die Orientierung von e_1 und e' ist irrelevant. Der Kreis \bar{C}_{T,e_2} geht von x über e_2 nach y (schwarze Kanten) zurück nach x über e' oder $\varphi(e')$ (rote Kanten). Im neuen Baum T' , der statt e' die Kante e_1 enthält, führt \bar{C}_{T',e_2}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

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

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++;
        }
    }
}
```

3 Implementierung

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 im letzten Schritt 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);}
}

```

3.3 Der Algorithmus

Mit diesen Datenstrukturen als Grundlage können wir nun den Algorithmus implementieren. 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>&));

```

3 Implementierung

```
//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 des Netzwerk-Simplex-Algorithmus 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 Codeauszügen 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` 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, indem wir uns in einer `std::map` für jeden Knoten merken, welcher Nachbar auf dem Weg zur Wurzel 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 (auto5 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 von einem der beiden Endknoten ausgeglichen 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.

3 Implementierung

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, 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. Für mich ist dies die kohärente Methode.

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 occurs

        //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 Exponentielle Instanzen nach Zadeh

Zadeh veröffentlichte 1973 in [10] eine Familie von Instanzen des Transportproblems, für die diverse verbreitete Algorithmen eine exponentielle Laufzeit benötigen. 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 für $j \geq 3$ rekursiv Netzwerke N_j mit jeweils $2j$ Knoten definieren, für die der umgesetzte Algorithmus mit HC und MaxVal $2^j + 2^{j-1} - 2$ Iterationen benötigt. Abb. 4.1 zeigt den Rekursionsanfang N_3 .

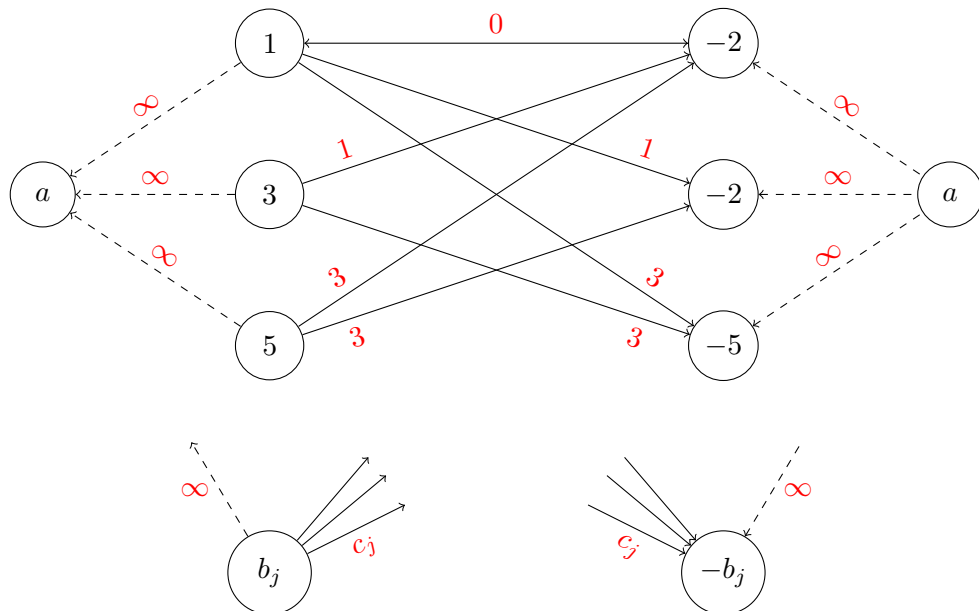


Abbildung 4.1: Grafik angelehnt an [10, S. 261]. Alle Kapazitäten sind unendlich, die Kosten wie angegeben. Die sechs mittleren Knoten bilden das Netzwerk N_3 , die gestrichelten Linien stellen den initialen aufspannenden Baum dar, wobei die beiden Knoten a identisch sein sollen. Die rekursive Erzeugung 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 = std::pow(2, 30); // "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. Die Quelle q_j verbinden wir mit jeder Senke $s_i \neq s_j$ über eine Kante (q_j, s_i) ; die Senke s_j ist von jeder Quelle $q_i \neq q_j$ aus über (q_i, s_j) erreichbar. Als Kosten für die neu hinzugefügten Kanten wählen wir $c((q_j, s_i)) = c((q_i, s_j)) := c_j = 2^{j-1} - 1$.

Zadeh legt in [10, S. 263 f.] dar, dass MaxVal-HC bei diesen Netzwerken in jeder nicht-degenerierten Iteration einen Kreis über den künstlichen Knoten um $\delta = 1$ augmentiert. Da nach Konstruktion der Durchfluss von N_j exponentiell steigt, ist die Anzahl an Iterationen ebenfalls exponentiell.

Wenn wir bereits ein **Network** n haben, das N_{j-1} entspricht, erhalten wir durch den folgenden Code N_j . Es wird davon ausgegangen, dass für eine Quelle q_i $\text{id} = 2*i$ und für eine Senke s_j $\text{id} = 2*i + 1$ gilt. Außerdem sollte u hinreichend groß sein.

```

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

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

```

In Abb. 4.2 ist die Laufzeit aller sechs eingeführten Varianten des Netzwerk-Simplex-Algorithmus auf den Netzwerken N_3 bis N_{10} dargestellt. Die Iterationsanzahl von MaxVal-HC steigt exponentiell, während die Version mit demselben Pivotalgorithmus und LC eine lineare Laufzeit aufweist. Die randomisierten Versionen scheinen stärker als linear zu wachsen, jedoch nicht exponentiell.

Damit lässt sich vermuten, dass Familien von exponentiellen Instanzen nicht nur einzeln für die jeweiligen Pivotalgorithmen, sondern zusätzlich differenziert nach der Initialisierung zu suchen sind. Im nächsten Kapitel beschäftigen wir uns mit algorithmischen Ansätzen für diese Suche.

4 Exponentielle Instanzen nach Zadeh

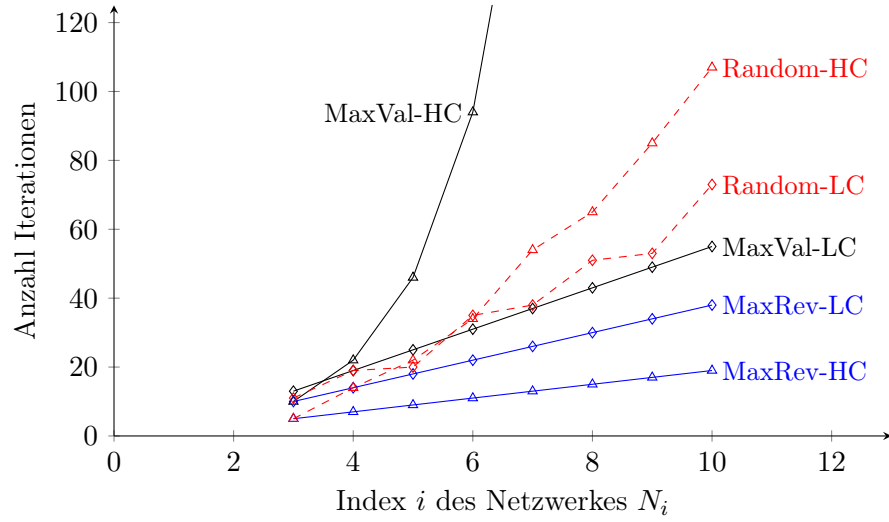


Abbildung 4.2: Anzahl der Iterationen der sechs eingeführten Algorithmusvarianten auf den Netzwerken N_i . Die Laufzeit von MaxVal-HC ist exponentiell, die der drei anderen deterministischen Varianten ist linear. Für den randomisierten Ansatz wurde der Schnitt von $|V(G_i)|$ Durchläufen genommen.

5 Experimentelle Ergebnisse

In Kapitel 2 wurde der Netzwerk-Simplex-Algorithmus mit drei verschiedenen Pivotalgorithmen und zwei Initialisierungen eingeführt. Es stellt sich nun die Frage, inwiefern sich die Laufzeiten dieser sechs Varianten nach oben oder unten abschätzen lassen. Für MaxVal-HC gibt es wie in Kapitel 4 beschrieben exponentielle Instanzen, wodurch wir die Möglichkeit haben, zumindest grob die Güte der nachfolgenden Ansätze zu bewerten: Findet eine von der konkreten Variante des Algorithmus unabhängige Routine exponentielle Instanzen für MaxVal-HC, so besteht die Hoffnung, dass diese für beliebige Pivotalgorithmen und Initialisierungen repräsentative Ergebnisse liefert. Leider ist keiner der implementierten Ansätze derart vielversprechend.

Es liegt nahe, zunächst zufällige Netzwerke zu betrachten. Der Konstruktor der Klasse `class RandomGraph` erzeugt zufällige Instanzen, die mit hoher Wahrscheinlichkeit zusammenhängend und lösbar sind. Die Untersuchung dieser Instanzen hat ergeben, dass die LC-Versionen zumeist mehr Iterationen benötigen als ihr HC-Pendant. MaxRev-HC und MaxVal-HC scheinen einander ebenbürtig zu sein, während Random konzeptionell bedingt auf Instanzen mit vielen Kanten und entsprechend vielen Kreisen eine schlechtere Laufzeit hat. In der Teilphase, in der MaxVal-LC auf den Kanten mit Kosten 0 iteriert, haben alle negativen Kreise $\bar{C}_{T,e}$ denselben Wert von $c(\bar{C}_{T,e}) = -2$. Damit verhält sich MaxVal-LC in dieser Teilphase ähnlich zu Random-LC. Schlechte Instanzen für Random-LC könnten dementsprechend auch bei MaxVal-LC zu einer hohen Iterationsanzahl führen.

In `class RandomGraph` sind weiterhin die beiden Funktionen `void evolve` und `void smartEvolve` enthalten. Erstere verändert ein gegebenes Netzwerk zufällig und speichert die Instanz mit der höchsten Iterationsanzahl, während die zweite stärker den evolutionären Ansatz betont und ihre Veränderungen umso wahrscheinlicher zurücknimmt, je kleiner die aktuelle Iterationsanzahl im Vergleich zur maximalen bisher gefundenen ist.

Sowohl `evolve` als auch `smartEvolve` beginnen mit einem vorher festgelegten Netzwerk – beispielsweise zufällig oder kantenlos gewählt – und bekommen eine Anzahl an Iterationen `size_t steps` übergeben. Pro Iteration wird eine der folgenden Aktionen ausgeführt:

- Ergänze den Graph um eine zufällig ausgewählte Kante.
- Lösche eine zufällig ausgewählte Kante aus dem Graph.
- Ändere die Kosten einer zufällig ausgewählten Kante.
- Ändere die Kapazität einer zufällig ausgewählten Kante.
- Ändere die b-Werte von zwei unterschiedlichen, zufällig ausgewählten Knoten um denselben Betrag mit entgegengesetzten Vorzeichen.

5 Experimentelle Ergebnisse

Alle oben aufgeführten Aktionen werden in dem `struct Action` gespeichert, das sich entweder die veränderte Kante oder die `id` der beiden modifizierten Knoten sowie deren `b`-Wert-Veränderung merkt:

```
struct Action {
    Action (Edge _oldE, Edge _newE) : oldE(_oldE), newE(_newE),
                                     edgeCase(true) {};
    Action (std::tuple<size_t, size_t, intmax_t> _b_change)
        : b_change(_b_change), edgeCase(false) {};

    Edge oldE, newE;
    std::tuple<size_t, size_t, intmax_t> b_change;
    bool edgeCase;
};
```

`smartEvolve` hat zusätzlich die Option, eine zufällig bestimmte Anzahl an Veränderungen zurückzunehmen. Die Wahrscheinlichkeitsverteilung über den vorhandenen Aktionen wurde entweder manuell oder zufällig gesetzt.

Anfänglich waren noch die Aktionen Knoten löschen bzw. hinzufügen enthalten, aber in den Versuchen hat sich das Hinzufügen eines Knoten nie bewährt. Bessere Ergebnisse erzielte eine Abfolge von Funktionsaufrufen, die jeweils mit der schlechtesten Instanz der vorherigen Iteration ergänzt um einen isolierten Knoten begannen.

Insgesamt gab es keine einzige Versuchsreihe, in der zumindest ein quadratisches Wachstum der Laufzeit beobachtet werden konnte. In vielen Fällen stieg die Anzahl an Iterationen nicht einmal linear zur Anzahl der Knoten. Um vernünftige Ergebnisse zu erzielen, sind anscheinend wesentlich ausgetüfteltere Ansätze notwendig.

6 Ausblick

In dieser Bachelorarbeit wurde der Netzwerk-Simplex-Algorithmus in Kapitel 2 graphentheoretisch hergeleitet und anschließend in Kapitel 3 dessen Implementierung dargestellt. Nachdem in Kapitel 4 die Existenz exponentieller Instanzen für eine der sechs Varianten verifiziert wurde, stand in Kapitel 5 die Frage im Raum, inwiefern sich für die anderen Versionen des Algorithmus obere oder untere Schranken zeigen lassen.

Die im Rahmen dieser Bachelorarbeit getesteten algorithmischen Ansätze konnten zu einer Antwort nicht viel beitragen. In einer Vertiefung dieser Thematik könnten komplexere evolutionäre Algorithmen oder andere Ansätze des maschinellen Lernens implementiert werden. Des Weiteren ist eine Erweiterung der Palette an Pivotalgorithmen oder Initialisierungen denkbar.

Sollte sich eine zuverlässige Routine zur Erzeugung schlechter Instanzen finden, kann diese genutzt werden, um Ansätze zu einer *Smoothed Analysis* im Sinne von [2] zu untersuchen. In der Klasse `Network` ist bereits eine von [11] inspirierte Funktion `void randomNoise(double phi)` enthalten, die zunächst für jede Kante $(v, w) \in E(G)$ zwei von `phi` und $\max_{e \in E(G)} \{u(e)\}$ abhängige Variablen ε_0 und ε_1 bestimmt. Daraufhin erhöht sie $b(v)$ um ε_0 , $b(w)$ um $-\varepsilon_0$ und $u((v, w))$ um $\varepsilon_0 + \varepsilon_1$. Bereits für sehr kleine Werte von `phi` und damit geringe Veränderungen weisen die Instanzen aus Kapitel 4 dann keine exponentielle Laufzeit mehr auf.

In einer praxisnäheren Fortführung dieser Arbeit würde die tatsächliche Rechenzeit an Stelle der Iterationsanzahl als Maß für die Laufzeit betrachtet werden und die Implementierung sowie Auswahl des Pivotalgorithmus diesbezüglich optimiert. Es wäre weiterhin interessant zu erfahren, ob die in Abschnitt 3.2 propagierte Beschleunigung durch die Klasse `Circle` auch zu beobachten ist, wenn pro Iteration nur ein Teil der Kreise berechnet wird.

Literaturverzeichnis

- [1] Mokhtar S. Bazaraa, John J. Jarvis und Hanif D. Sherali. „Linear Programming and Network Flows“. In: 4. Aufl. Hoboken, New Jersey: John Wiley & Sons, Inc., 2010, 453 ff. ISBN: 978-0-470-46272-0.
- [2] Tobias Brunsch u. a. „Smoothed Analysis of the Successive Shortest Path Algorithm“. In: *CoRR* abs/1501.05493 (2015). arXiv: 1501.05493. URL: <http://arxiv.org/abs/1501.05493>.
- [3] Vašek Chvátal. „Linear Programming“. In: 16. Aufl. Series of books in the mathematical sciences. W. H. Freeman, 2002, 291 ff. ISBN: 978-0-7167-1587-0.
- [4] W. H. Cunningham. „A network simplex method“. In: *Mathematical Programming* 11.1 (1976), S. 105–116. ISSN: 1436-4646. DOI: 10.1007/BF01580379.
- [5] G. B. Dantzig. „Application of the simplex method to a transportation problem“. In: *Activity Analysis of Production and Allocation*. Hrsg. von T. C. Koopmans. New York: Wiley, 1951. Kap. XXIII, S. 359–373.
- [6] Stefan Hougardy und Jens Vygen. *Algorithmische Mathematik*. Springer-Lehrbuch. Berlin/Heidelberg: Springer Spektrum, 2015. ISBN: 978-3-662-47013-8.
- [7] Dieter Jungnickel. *Graphs, Networks and Algorithms*. 4. Aufl. Algorithms and Computation in Mathematics 5. Heidelberg u. a.: Springer, 2013. ISBN: 978-3-642-32277-8.
- [8] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. 1. Aufl. Addison-Wesley Professional, 2001. ISBN: 978-0-201-74962-5.
- [9] Alex Orden. „The Transshipment Problem“. In: *Management Science* 2.3 (1956), S. 276–285. ISSN: 00251909, 15265501. URL: <http://www.jstor.org/stable/2627138>.
- [10] Norman Zadeh. „A bad network problem for the simplex method and other minimum cost flow algorithms“. In: *Mathematical Programming* 5.1 (1973), S. 255–266. ISSN: 1436-4646. DOI: 10.1007/BF01580132.
- [11] Xianghui Zhong. Persönliche Mitteilung. 2018.

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Unterschrift