Preflow-Push Algorithmen

Nicolai Hähnle*

11. April 2006

Inhaltsverzeichnis

1	Einführung	1
2	Formale Problembeschreibung	2
3	Rest-Netzwerke	4
4	Preflows und die Höhenfunktion	5
5	Der generische Preflow-Push Algorithmus	7
6	Laufzeitanalyse	10
7	Der FIFO Preflow-Push Algorithmus	14
8	Der Highest-Label Preflow-Push Algorithmus	16
9	Zusammenfassung und Ausblick	16

1 Einführung

Stellen Sie sich ein Netzwerk aus Umspannwerken und Hochspannungsleitungen vor, über das ein Kraftwerk eine entfernte Stadt mit Energie versorgen soll. Natürlich ist die Kapazität jeder einzelnen Leitung beschränkt. Welche Leistung kann maximal über das Netzwerk transportiert werden? Welche Leistung kann man noch erreichen, wenn bestimmte Leitungen ausfallen? Solche und ähnliche Fragen können mit Hilfe von Preflow-Push Algorithmen schnell beantwortet werden. Preflow-Push Algorithmen werden benutzt, um offline eine maximal mögliche Ausnutzung des Netzwerks zu berechnen, die dann nach Abschluss der Berechnung in der Realität umgesetzt werden kann.

Ich orientiere mich hauptsächlich an [Ahu]. Preflow-Push Algorithmen werden auch in [CLRS] vorgestellt, allerdings werden sie dort „Push-Relabel algorithms“ genannt. Die in der Literatur verwendeten Notationen und Begriffe sind leider nicht einheitlich. Die von mir genutzte Notation basiert auf [Ahu] mit vereinzelten Anleihen aus [CLRS].

*prefect@upb.de

2 Formale Problembeschreibung

Im Folgenden werde ich die intuitive Vorstellung von Netzwerken, in denen zum Beispiel Strom durch Leitungen und Umspannwerke oder Wasser durch Rohre und Pumpen geleitet wird, formalisieren.

Definition 2.1. *Ein* Fluss-Netzwerk $N = (G, c, s, t)$ *ist ein gerichteter Graph* $G = (V, E)$ *mit einer* Kapazitätsfunktion $c : V \times V \to \mathbb{R}$, *einer* Quelle $s \in V$ *und einer* Senke $t \in V$, *so dass gilt:*

a) $s \neq t$

b) $c(v, w) \geq 0$ *für alle* $v, w \in V$

c) $c(v, w) > 0$ *genau dann, wenn* $(v, w) \in E$

Die Kantenmenge des Graphen enthält genau die Kanten, die für die folgenden Überlegungen relevant sind, das heißt die Kanten, entlang denen tatsächlich Kapazität vorhanden ist. Der zugrunde liegende Graph ist gerichtet. Um auszudrücken, dass zwischen zwei Knoten prinzipiell in beide Richtungen Fluss gepumpt werden kann, müssen Kanten in beide Richtungen in den Graphen eingefügt werden.

Im Folgenden setze ich, sofern nicht anders angegeben, stets ein Fluss-Netzwerk $N = (G, c, s, t)$, $G = (V, E)$ voraus.

Der Transport von Gütern durch ein Fluss-Netzwerk wird durch eine zusätzliche Beschriftung der Kanten modelliert. Die Beschriftung einer Kante gibt dabei an, wieviel (stets nichtnegativer) Fluss in Richtung einer Kante fließt.

Definition 2.2. *Ein* Quasi-Fluss *ist eine Funktion* $x : V \times V \to \mathbb{R}$ *mit* $0 \leq x(v, w) \leq c(v, w)$ *für alle* $v, w \in V$.

Ein so definierter Quasi-Fluss überschreitet niemals die Transportkapazität eines Netzwerks. Allerdings kann es vorkommen, dass in einen Knoten mehr hinein- als hinausfließt, oder umgekehrt. Wir wollen aber alle Knoten außer Quelle und Senke für das sogenannte Maximum-Flow Problem lediglich wie Pumpstationen betrachten und daher diese Situation der Unausgewogenheit in einem Knoten ausschließen.

Definition 2.3. *Der* Überschuss *(engl.* excess*) in einem Knoten* $v \in V$ *bezüglich Fluss* x *ist definiert durch:*

$$e_x(v) := \sum_{u \in V} x(u, v) - \sum_{w \in V} x(v, w)$$

Ist der Fluss x *aus dem Kontext klar, schreibe ich auch einfach* $e(v)$ *statt* $e_x(v)$.

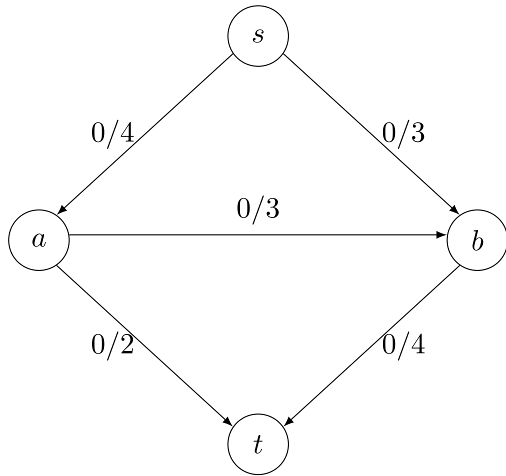
Definition 2.4. *Ein* Fluss $x : V \times V \to \mathbb{R}$ *ist ein Quasi-Fluss mit:*

$$e_x(v) = 0 \quad \textit{für alle } v \in V \setminus \{s, t\}$$

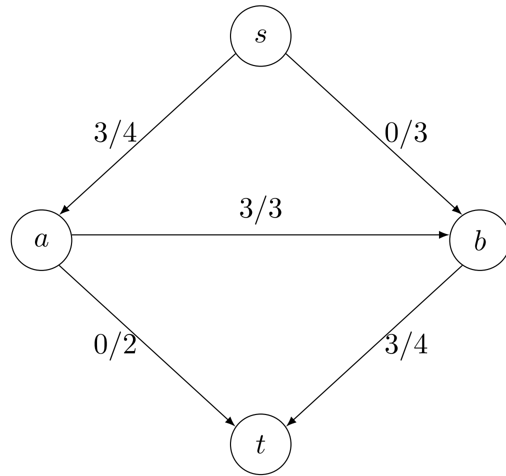
Der Wert *eines Flusses ist* $v_x := e(t)$.

Die Bedingung an den Fluss erzwingt die Ausgewogenheit in allen Knoten außer der Quelle und der Senke. Die Aufgabenstellung kann jetzt formal wie folgt definiert werden.

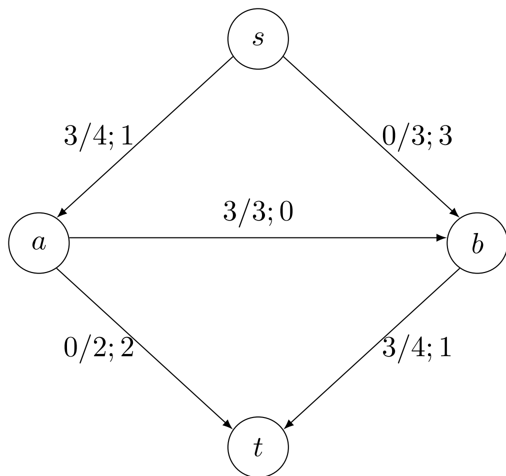
Maximum-Flow-Problem. Gegeben ein Fluss-Netzwerk $N = (G, c, s, t)$, finde einen zulässigen Fluss x in N mit maximalem Wert, also einen *maximalen Fluss.*



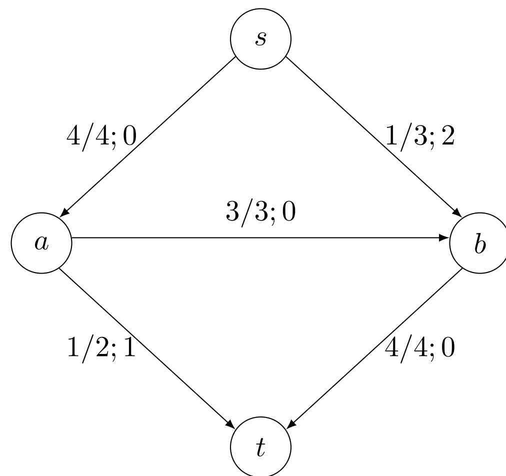
(a) Ein Fluss-Netzwerk...



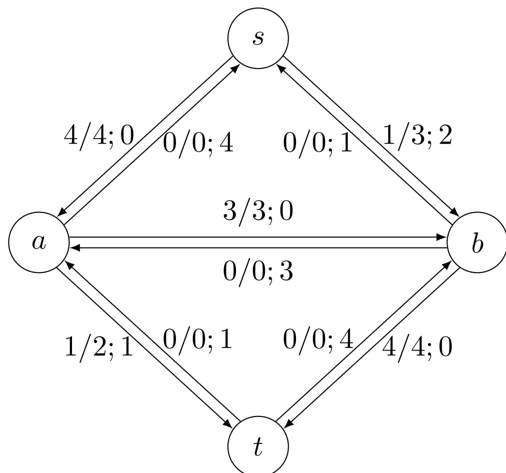
(b) ... mit einem einfachen Fluss



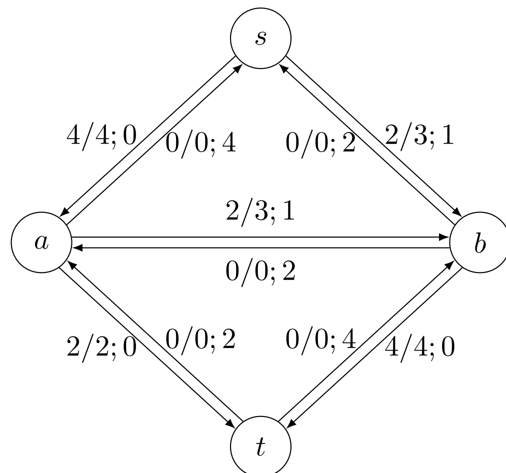
(c) Der erste Ansatz zur Berechnung der Reste...



(d) ... führt in eine Sackgasse



(e) Die korrigierte Berechnung der Reste



(f) Ein maximaler Fluss

Abbildung 1: Herleitung der Berechnung von Restkapazitäten. Kanten sind mit $x/c; r$ beschriftet. Dabei ist x der Fluss, c die ursprüngliche Kapazität und r die Kapazität im Restnetzwerk.

3 Rest-Netzwerke

Es gibt verschiedene Möglichkeiten, an das Maximum-Flow-Problem heranzugehen, aber in jedem Fall wird man versuchen, sich sukzessive einer Lösung zu nähern. Betrachten wir einmal das recht einfache Fluss-Netzwerk in Abb. 1(a). Nehmen wir an, ein Lösungsalgorithmus leitet in diesem Beispiel 3 Einheiten von s über a und b nach t. Der daraus resultierende Fluss ist in Abb. 1(b) gezeigt.

Damit ein Algorithmus ausgehend von diesem Fluss bessere Lösungen berechnen kann, muss er die *verbleibenden* Kapazitäten im Netzwerk berücksichtigen. Diese verbleibenden Kapazitäten werden durch ein Rest-Netzwerk modelliert. Zunächst aber eine kleine Vorüberlegung. In Abb. 1(c) wurden Restkapazitäten eingetragen, die gemäß der Formel $r(v, w) = c(v, w) - x(v, w)$ berechnet wurden, also einfach ursprüngliche minus im Fluss genutzte Kapazität. Wenn wir jetzt den Fluss weiter erhöhen, uns aber immer auf die eingezeichneten Restkapazitäten beschränken, erreichen wir maximal den in Abb. 1(d) gezeigten Fluss vom Wert 5, in dem entlang der Pfade (s, a, t) und (s, b, t) jeweils eine Einheit zusätzlich geschickt wird.

In Wirklichkeit gibt es aber für das ursprüngliche Problem einen maximalen Fluss vom Wert 6, der in Abb. 1(f) gezeigt wird. Im Beispiel hat bereits der erste Schritt, nämlich das Erzeugen von 3 Einheiten Fluss entlang des Pfads (s, a, b, t) in eine vermeintliche Sackgasse geführt. Der naive gierige Ansatz zur Berechnung der Reste führt also zu falschen Ergebnissen.

Eine frühere falsche Entscheidung kann aber sehr elegant nachträglich rückgängig werden. Dazu berücksichtigen wir bei der Berechnung der Restkapazitäten, dass ein bereits erzeugter Fluss entlang einer Kante einfach wieder entfernt werden kann. Die Restkapazität berechnet sich dann als $r(v, w) = c(v, w) - x(v, w) + x(w, v)$. Die so aus Abb. 1(d) errechneten Reste sind in Abb. 1(e) gezeigt. Insbesondere ist zu beachten, dass hier auch Kanten in umgekehrter Richtung auftreten.

An diesen auf verbesserte Weise ausgerechneten Resten kann man sehen, dass entlang des Pfades (s, b, a, t) noch die Restkapazität 1 vorhanden ist. Nutzt man diese Restkapazität aus, gelangt man zum maximalen Fluss, der in Abb. 1(f) gezeigt ist. Dabei wird entlang der Kante (b, a) Fluss „erzeugt“, indem der Fluss in Gegenrichtung reduziert wird.[1]

Dies motiviert die folgende Definition des Rest-Netzwerks.

Definition 3.1. *Sei $x : V \times V \to \mathbb{R}$ ein Quasi-Fluss im Fluss-Netzwerk $N = (G, c, s, t)$, $G = (V, E)$. Dann ist das Rest-Netzwerk $R(x)$ definiert durch*

$$R(x) := (G(x), r_x, s, t), \quad G(x) := (V, E(x))$$

wobei gilt:

a) $r_x(v, w) := c(v, w) - x(v, w) + x(w, v)$ *für alle* $v, w \in V$

b) $(v, w) \in E(x)$ *genau dann, wenn* $r_x(v, w) > 0$

Wenn x aus dem Kontext klar ist, schreibe ich auch $r(v, w) = r_x(v, w)$

[1] Da der maximale Fluss *offline* berechnet wird ist physikalisch noch nichts passiert. Man kann also so tun, als ob ein Entfernen von bereits erzeugtem Fluss dasselbe ist wie ein Hinzufügen von Fluss in die Gegenrichtung, auch wenn im ursprünglichen Graphen keine entsprechend gerichtete Kante vorhanden ist.

Beachte, dass bezüglich des maximalen Flusses aus Abb. 1(f) kein Pfad von der Quelle zur Senke existiert, auf dem noch Restkapazität vorhanden ist. Anschaulich ist das plausibel: Wenn es einen solchen Pfad gäbe, könnte man entlang dieses Pfades den Fluss erhöhen, der ursprüngliche Fluss wäre also gar nicht maximal gewesen. Der folgende Satz besagt, dass diese Beobachtung von allgemeiner Bedeutung ist.

Satz 3.2 (Augmenting Path Theorem). *Ein Fluss ist genau dann maximal, wenn im Rest-Netzwerk kein Pfad von der Quelle zur Senke existiert.*

Einen Beweis dieses Satzes finden Sie auf S. 657 von [CLRS].

4 Preflows und die Höhenfunktion

Die Grundidee von Preflow-Push Algorithmen ist, zunächst so viel Fluss wie möglich von der Quelle in benachbarte Knoten zu pumpen, und danach den Fluss immer nur stückweise entlang einzelner Kanten „weiterzupushen“. In Abb. 2 sind die ersten Schritte einer möglichen Ausführung des Preflow-Push Algorithmus gezeigt.

Bei dieser Vorgehensweise erhält man in den Zwischenschritten natürlich keinen Fluss, da die Bedingung $e(v) = 0$ für $v \in V \setminus \{s, t\}$ verletzt wird. Stattdessen arbeiten Preflow-Push Algorithmen mit sogenannten Preflows.

Definition 4.1. *Ein Quasi-Fluss* $x : V \times V \to \mathbb{R}$ *heißt* Preflow*, wenn gilt:*

$$e_x(v) \geq 0 \quad \text{für alle } v \in V \setminus \{s, t\}$$

Ein Knoten $v \in V \setminus \{s, t\}$ *heißt* aktiv*, wenn* $e_x(v) > 0$ *gilt.*

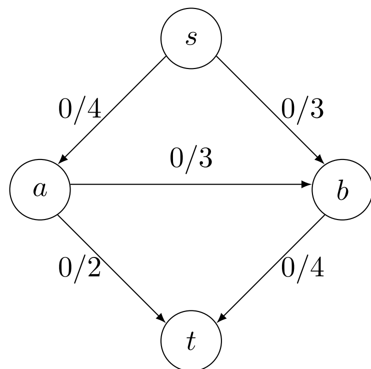
Außerdem stellt sich die Frage, entlang welcher der im Rest-Netzwerk ausgehenden Kanten eines aktiven Knotens der Überschuss geschickt werden soll. Schließlich soll der Überschuss nicht zu früh in die Quelle zurückfliessen. Stellen wir uns vor, durch das Flussnetzwerk soll Wasser geschickt werden. Wasser fließt für gewöhnlich abwärts, also stellen wir uns weiter vor, dass sich die Quelle auf einem Berg und die Senke in einem Tal befindet. Die Höhe der anderen Knoten ist variabel: Sammelt sich das Wasser an einem Knoten ohne eine Möglichkeit, weiter abzufließen, so wird dieser Knoten einfach angehoben. Diese Metapher von Wasserleitungen und Knoten variabler Höhe kann sehr hilfreich sein, um den Preflow-Push Algorithmus zu verstehen, auch wenn die Analogie in einigen Details zusammenbricht. In jedem Fall motiviert sie die folgende Definition.

Definition 4.2. *Die* Höhenfunktion $h : V \to \mathbb{N}_0$ *ordnet jedem Knoten* v *die* Höhe $h(v)$ *zu. Eine Kante* (v, w) *heißt* zulässig*, wenn*

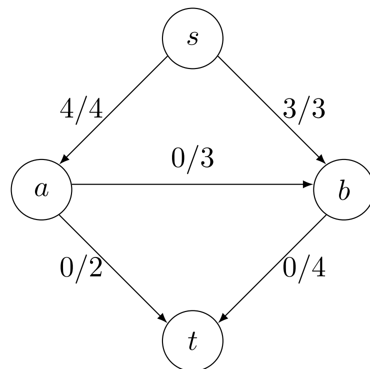
a) $(v, w) \in E(x)$ *(d.h. entlang* (v, w) *existiert Restkapazität) und*

b) $h(v) > h(w)$

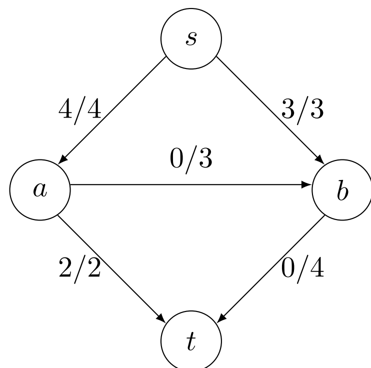
Zulässige Kanten sind also immer abwärts gerichtet. Wenn von einem aktiven Knoten $v \in V$ keine zulässige Kante mehr ausgeht, wird der Knoten angehoben, also $h(v)$ vergrößert, bis wieder eine zulässige Kante zur Verfügung steht. In Abb. 3 wird die Höhenfunktion graphisch angedeutet für den Preflow aus Abb. 2(e). Der Überschuss von b müsste in die Quelle zurückfliessen, doch die Quelle liegt zu weit oben, weshalb b im nächsten Schritt um mindestens eine Einheit angehoben werden muss.



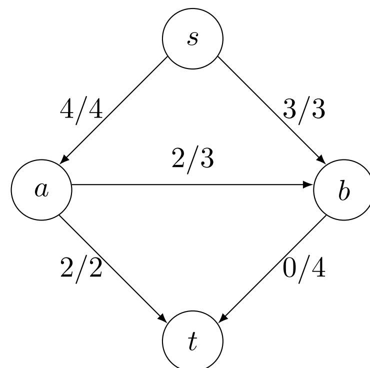
(a) Ein Fluss-Netzwerk



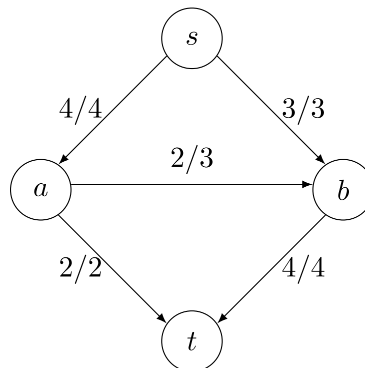
(b) Nach der Initialisierung sind a und b aktiv



(c) Pushen von a nach t



(d) Pushen von a nach b



(e) Pushen von b nach t

Abbildung 2: Erste Schritte von Preflow-Push. Kanten sind mit x/c beschriftet. Dabei ist x der Fluss und c die ursprüngliche Kapazität. In (e) ist bereits ein maximaler Fluss in die Senke erreicht, aber der Überschuss von b muss noch in die Quelle zurückfliessen.

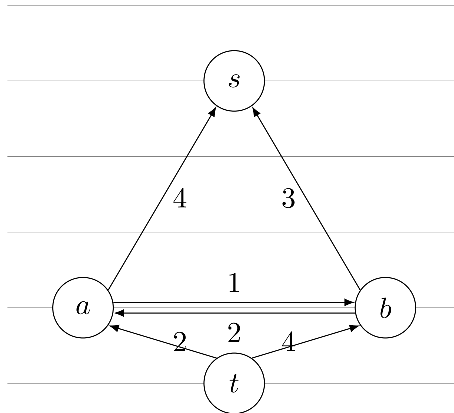


Abbildung 3: Das Rest-Netzwerk zu Abb. 2(e) mit durch Höhenlinien angedeuteter Höhenfunktion

5 Der generische Preflow-Push Algorithmus

Die Grundidee aus dem letzten Abschnitt kann wie folgt als Algorithmus umgesetzt werden. Ich verzichte im Pseudocode darauf, jede einzelne Variablen-Zuweisung anzugeben. Unter anderem die Kapazitäten im Rest-Netzwerk und die Überschüsse in den Knoten werden bei Bedarf neu berechnet, auch wenn dies nicht explizit im Pseudocode angegeben ist.

Preflow-Push()

1 Preprocess()
2 **while** $\exists$ aktiver Knoten $v \in V$
3   **do if** $\exists$ zulässige Kante (v, w)
4   **then** Push(v, w)
5   **else** Relabel(v)

In welcher Reihenfolge aktive Knoten und zulässige Kanten in den Zeilen 2 bzw. 3 ausgewählt werden ist hier offen gelassen, weshalb vom *generischen* Preflow-Push Algorithmus gesprochen wird.

Preprocess()

6 $x \leftarrow 0$
7 **for** alle Kanten $(s, w) \in E$
8   **do** $x(s, w) \leftarrow c(s, w)$
9 **for** alle Knoten $v \in V \setminus \{s\}$
10   **do** $h(v) \leftarrow 0$
11 $h(s) \leftarrow |V|$

Push(v, w)

12 $\delta \leftarrow \min\{e(v), r(v, w)\}$
13 Erhöhe Fluss entlang (v, w) um δ Einheiten

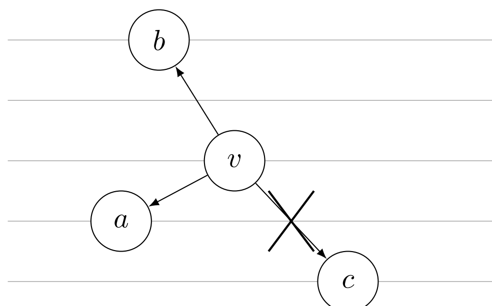


Abbildung 4: Die Kante (v, c) verletzt I3, die anderen Kanten erfüllen die Bedingung der Invariante.

Das Pumpen von Fluss wird *Push* genannt. Genauer spricht man von einem *sättigenden* Push, falls $\delta = r(v, w)$ (die Kante (v, w) wird gesättigt), und andernfalls von einem *nichtsättigenden* Push. Nach einem nichtsättigenden Push ist der Überschuss $e(v) = 0$, der Knoten also nicht mehr aktiv. Nach einem sättigenden Push bleibt der Knoten v dagegen im Allgemeinen aktiv.

Relabel(v)
14 $\quad h(v) \leftarrow \min\{h(w) + 1 \mid (v, w) \in E(x)\}$

Das Erhöhen von $h(v)$ in Zeile 14 wird *Relabel-Operation* genannt. Dass die Menge, über die das Minimum gebildet wird, nicht leer sein kann folgt aus Lemma 5.2.

Um die Korrektheit des Algorithmus zu zeigen betrachte ich die folgenden Invarianten.

I1) x ist ein Preflow.

I2) $h(s) = |V|$ und $h(t) = 0$

I3) Falls $(v, w) \in E(x)$ eine Kante im Rest-Netzwerk ist, so gilt $h(v) \leq h(w) + 1$

Die Invariante I3 bedeutet anschaulich, dass die Höhe eines Knotens nicht beliebig ansteigen kann, sondern durch die Nachbarn des Knotens im Rest-Netzwerk beschränkt wird. Knoten werden durch Kanten im Rest-Netzwerk also gewissermaßen verankert. Abb. 4 zeigt dies an einem Beispiel-Ausschnitt aus einem Rest-Netzwerk.

Lemma 5.1. *Nach Aufruf von* Preprocess *gelten die Invarianten I1 bis I3.*

Beweis. Für alle $v, w \in V$ gilt nach den Zeilen 6-8 $x(v, w) = 0$ oder $x(v, w) = c(v, w)$. Die Kapazität des Fluss-Netzwerks wird also nicht überschritten. Der einzige Knoten, von dem positive Flüsse ausgehen, ist die Quelle. Demnach gilt für alle anderen Knoten $e(v) \geq 0$. Damit ist I1 gezeigt.

I2 ist klar, die Höhen werden gerade entsprechend zugewiesen.

Sei $(v, w) \in E(x)$ beliebig. Da alle von der Quelle ausgehenden Kanten mit maximalem Fluss belegt wurden, ist keine dieser Kanten im Rest-Netzwerk vorhanden. Daraus folgt $v \neq s$. Falls $w = s$, so ist $h(v) = 0 < |V| = h(w) < h(w) + 1$. Im Fall $w \neq s$ ist $h(v) = 0 = h(w) < h(w) + 1$. In jedem Fall gilt $h(v) \leq h(w) + 1$, und damit ist I3 gezeigt. □

Lemma 5.2. *Sei $v \in V$ ein aktiver Knoten. Dann existiert im Rest-Netzwerk ein (gerichteter) Pfad von v zur Quelle.*

Beweisidee. Sei v ein aktiver Knoten, das heißt der Knoten v hat einen positiven Überschuss. Verfolgt man schrittweise zurück, von wo dieser Überschuss herkommen kann, so entsteht ein Pfad im Rest-Netzwerk, der letztendlich einen Knoten w mit $e(w) < 0$ erreichen muss. Da x ein Preflow ist, ist der einzige Knoten mit $e(w) < 0$ die Quelle. Also gibt es im Rest-Netzwerk einen Pfad von v zur Quelle.

Formale Beweise findet man in [Ahu] (Lemma 7.11) und [CLRS] (Lemma 26.20). □

Lemma 5.3. *Der Aufruf von* Push*(v, w) in Zeile 4 erhält die Invarianten I1 bis I3.*

Beweis. Der Push vergrößert $e(w)$ und verringert $e(v)$. Durch die Wahl von δ wird $e(v)$ aber höchstens auf 0 verringert. Damit ist x auch nach dem Aufruf von Push ein Preflow, I1 bleibt also erhalten. Da die Höhenfunktion durch Push nicht verändert wird, bleibt I2 erhalten.

Bleibt noch zu zeigen, dass auch I3 erhalten bleibt. Durch einen Push von v nach w können zwei Änderungen an der Kantenmenge des Rest-Netzwerks ausgelöst werden. Zum einen kann die Kante (v, w) (durch einen sättigenden Push) entfernt werden. Dies ist für I3 unerheblich. Zum anderen kann die Kante (w, v) zum Rest-Netzwerk hinzugefügt werden. Da die Kante (v, w) zulässig war (sonst wäre Push nicht aufgerufen worden), gilt $h(w) < h(v) < h(v) + 1$. Damit ist I3 erfüllt. □

Lemma 5.4. *Der Aufruf von* Relabel*(v) in Zeile 5 erhält die Invarianten I1 bis I3 und hebt v um mindestens eine Einheit an.*

Beweis. Nach Definition sind Quelle und Senke niemals aktiv, also ändern sich auch die Höhen dieser Knoten nicht. I2 bleibt also erhalten. Außerdem ändert Relabel den Preflow nicht, damit bleibt auch I1 erhalten.

Im Folgenden bezeichne ich mit h die Höhenfunktion vor der Relabel-Operation und mit h' die Höhenfunktion danach. Da von v keine zulässige Kante ausging, gilt für alle Kanten $(v, w) \in E(x)$: $h(w) \geq h(v)$. Da nun $h'(v)$ um eine Einheit größer ist als das kleinste dieser $h(w)$ gilt $h'(v) \geq h(v) + 1$. Es bleibt nur noch zu zeigen, dass I3 erhalten wird.

Durch Relabel ändert sich die Kantenmenge im Rest-Netzwerk nicht. Es ist also lediglich nachzuprüfen, ob I3 für alle in v hinein- und aus v hinausführenden Kanten erfüllt ist. Sei $(v, w) \in E(x)$ eine hinausführende Kante. Damit ist nach Zeile 14:

$$h'(v) = \min_{(v,u) \in E(x)} (h'(u) + 1) \leq h'(w) + 1$$

Sei nun $(w, v) \in E(x)$ eine hineinführende Kante. Da I3 vor dem Aufruf erfüllt war, gilt

$$h'(w) = h(w) \leq h(v) + 1 < h'(v) + 1$$

da die Höhe von v nur vergrößert werden kann. Damit ist I3 weiterhin erfüllt. □

Lemma 5.5. *Im berechneten Rest-Netzwerk $N(x)$ existiert kein Pfad von s nach t.*

Beweis. Angenommen, es gibt einen einfachen Pfad P (ohne Zyklen) von s nach t im Rest-Netzwerk. Schreibe P als

$$P = (s, v_1, \ldots, v_{k-1}, t)$$

wobei k die Länge des Pfades ist. Dann gilt wegen I3 und I2:

$$|V| = h(s) \leq h(v_1) + 1 \leq ... \leq h(v_{k-1}) + k - 1 \leq h(t) + k = k$$

Der Pfad hat also mindestens die Länge $|V|$, aber jeder kreisfreie Pfad hat grundsätzlich höchstens Länge $|V| - 1$. Dieser Widerspruch zeigt die Behauptung. □

Satz 5.6. *Nach Beendigung von* PREFLOW-PUSH *ist x ein maximaler Fluss.*

Beweis. Nach Beendigung des Algorithmus gibt es keine aktiven Knoten, d.h. $e(v) = 0$ für alle $v \in V \setminus \{s, t\}$. Der berechnete Preflow x ist also ein Fluss. Nach Lemma 5.5 gibt es außerdem keinen Pfad von s nach t im Rest-Netzwerk. Nach Satz 3.2 (Augmenting Path Theorem) ist der berechnete Fluss also maximal. □

Satz 5.6 besagt, dass der generische Preflow-Push Algorithmus einen maximalen Fluss berechnet hat, wenn er terminiert. Damit ist jedoch noch nicht gezeigt, dass der Algorithmus auch tatsächlich bei jeder Eingabe terminiert. Dies werde ich im nächsten Abschnitt nachholen, indem ich eine obere Schranke für die Laufzeit des Algorithmus zeige.

6 Laufzeitanalyse

Ein erstes wichtiges Resultat, das im Verlauf der Laufzeitanalyse immer wieder benötigt wird, ist folgende Schranke für die Höhen von Knoten.

Lemma 6.1. *Zu jeder Zeit während der Ausführung von* PREFLOW-PUSH *gilt $h(v) < 2|V|$ für alle $v \in V$.*

Beweis. Für $v = s$ und $v = t$ folgt die Behauptung aus I2. Sei also $v \in V \setminus \{s, t\}$. Falls v noch nie relabelt wurde gilt auf Grund von PREPROCESS $h(v) = 0 < 2|V|$.

Betrachte also den Fall, dass v mindestens einmal relabelt wurde. Direkt nach der letzten Relabel-Operation auf v war der Knoten v noch aktiv. Nach Lemma 5.2 gab es zu diesem Zeitpunkt einen Pfad P der Länge $k < |V|$ von v nach s. Sei

$$P = (v, v_1, v_2, \ldots, v_{k-1}, s)$$

Wenn h' die Höhenfunktion nach der letzten Relabel-Operation auf v ist, so gilt:

$$h'(v) \leq h'(v_1) + 1 \leq h'(v_2) + 2 \leq \ldots \leq h'(s) + k < h'(s) + |V| = 2|V|$$

Da die Höhe von v nur durch eine Relabel-Operation auf v geändert werden kann ist $h(v) = h'(v) < 2|V|$. □

Vor der eigentlichen Worst-Case-Laufzeitanalyse will ich wichtige Details der verwendeten Datenstrukturen klären. Die Kanten des Graphen werden in Adjazenzlisten gespeichert. In den formalen Überlegungen wurde eine Kante immer nur dann in den entsprechenden Graphen aufgenommen, wenn die Kapazität entlang dieser Kante positiv war. Für die Laufzeitanalyse gehe ich jedoch davon aus, dass der Algorithmus die Adazenzlisten im Speicher nicht ändert. Eine Kante bleibt also auch dann in der entsprechenden Adjazenzliste des Graphen, wenn ihre Kapazität 0 wird. Insbesondere wird auch die Reihenfolge der Kanten innerhalb von Adjazenzlisten nie geändert. Stattdessen

soll der Algorithmus wenn nötig implizit die Kapazität von Kanten prüfen. Dadurch bleibt die Korrektheit des Algorithmus erhalten. Die Prüfung benötigt bei Speicherung der benötigten Daten nur konstante Laufzeit, wirkt sich also auf das asymptotische Laufzeitverhalten des Algorithmus nicht aus.

Außerdem ist der gespeicherte Graph symmetrisch, d.h. wenn die Kante (v, w) in der Adjazenzliste von v steht, dann steht (w, v) in der Adjazenzliste von w. Die Größe der so entstandenen Kantenmenge bezeichne ich im Folgenden mit $|E|$. Sie ist höchstens doppelt so groß wie die ursprüngliche Kantenmenge. Dieser konstante Faktor ist für die Untersuchung des asymptotischen Laufzeitverhaltens unwichtig. Ich nehme außerdem an, dass der Graph zusammenhängend ist. Diese Annahme ist keine Einschränkung, da nur die Knoten, die von der Quelle aus erreichbar sind, auch zum Fluss beitragen können. Andere Knoten könnten daher vor der eigentlichen Ausführung bzw. im Initialisierungsschritt entfernt werden.[2]

Ferner werden die aktiven Knoten in einer Linked List gespeichert, so dass aktive Knoten in konstanter Zeit gefunden, eingefügt und entfernt werden können.

Das Suchen von zulässigen Kanten in Zeile 3 von PREFLOW-PUSH bedarf genauerer Überlegungen. Damit nicht jedes Mal die gesamte Adjazenzliste eines Knotens durchlaufen werden muss, wird für jeden Knoten v ein Zeiger $current(v)$ eingeführt, der auf eine Kante (v, w) der Adjazenzliste des Knotens zeigt. Dieser *current*-Zeiger wird nach den folgenden Regeln verwendet:

1. PREPROCESS initialisiert den *current*-Zeiger jedes Knotens auf die erste Kante der Adjazenzliste.

2. RELABEL(v) setzt den *current*-Zeiger von v auf die erste Kante der Adjazenzliste zurück.

3. Falls der *current*-Zeiger bei der Suche nach zulässigen Kanten in Zeile 3 von PREFLOW-PUSH nicht auf eine zulässige Kante zeigt, wird der Zeiger so lange entsprechend der Adjazenzliste weitergeschoben, bis eine zulässige Kante gefunden wurde.

4. Erreicht Zeile 3 auf diese Weise das Ende der Adjazenzliste, so wird RELABEL aufgerufen.

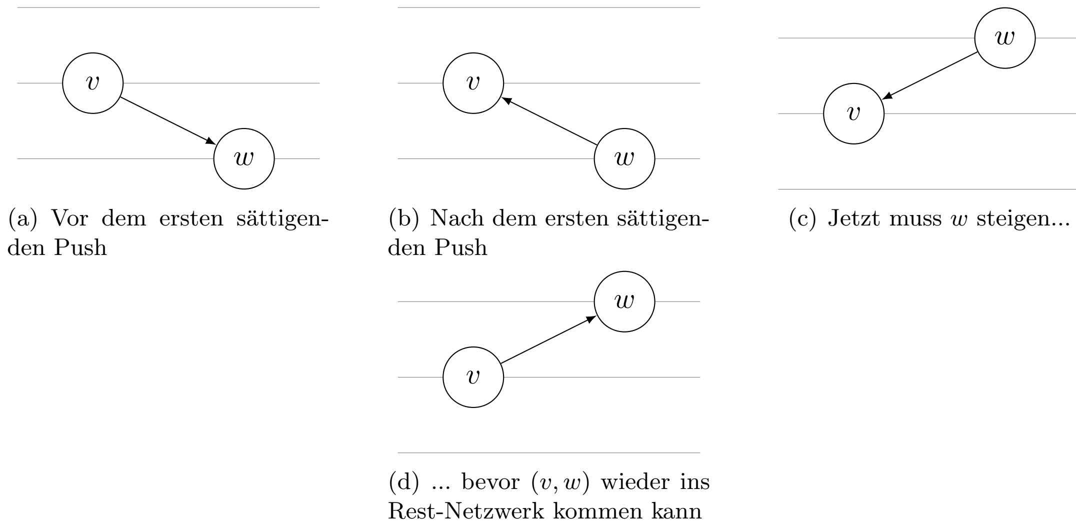
Das folgende Lemma besagt, dass eine bei der Suche nach zulässigen Kanten einmal verworfene Kante erst durch eine Relabel-Operation wieder zulässig werden kann. Beim Relabeln wird aber der entsprechende *current*-Zeiger auf den Anfang der Adjazenzliste zurückgesetzt, so dass die Korrektheit des geschilderten Verfahrens garantiert ist.

Lemma 6.2. *Eine unzulässige Kante (v, w) (dabei darf $(v, w) \notin E(x)$ gelten) kann nur durch eine Relabel-Operation auf v zulässig werden.*

Beweis. Sei (v, w) eine unzulässige Kante. Es sind zwei Fälle zu unterscheiden. Im ersten Fall gilt $(v, w) \notin E(x)$, das heißt $r(v, w) = 0$. Bevor die Kante (v, w) zulässig werden kann, muss also entlang (w, v) gepusht werden. Dazu muss aber zuerst (w, v) zulässig werden, und dafür muss $h(w) > h(v)$ gelten. Nach einem Push entlang (w, v) ist (v, w) also nach wie vor unzulässig, dann aber entsprechend dem zweiten Fall.

Im zweiten Fall gilt $h(v) < h(w)$. Da die Höhe von Knoten nur steigen kann muss offensichtlich v relabelt werden, bevor (v, w) zulässig werden kann. □

[2] Dies kann in linearer Zeit zum Beispiel durch eine Breitensuche geschehen und beeinträchtigt daher nicht die asymptotische Laufzeit des Algorithmus.



(a) Vor dem ersten sättigenden Push

(b) Nach dem ersten sättigenden Push

(c) Jetzt muss w steigen...

(d) ... bevor (v, w) wieder ins Rest-Netzwerk kommen kann

Abbildung 5: Die Beweisidee zu Lemma 6.4: Zwischen zwei sättigenden Pushes entlang von (v, w) muss w um mindestens 2 Einheiten steigen.

Lemma 6.3. *Der Algorithmus benötigt für Relabel-Operationen und zum Suchen von zulässigen Kanten insgesamt Zeit* $\mathcal{O}(|V||E|)$.

Beweis. Betrachte einen festen Knoten $v \in V$. Da die Höhe eines Knotens mit jeder Relabel-Operation um mindestens eine Einheit zunimmt, die Höhe aber gleichzeitig durch $2|V|$ beschränkt ist, wird jeder einzelne Knoten $\mathcal{O}(|V|)$ mal relabelt.

Zwischen je zwei Relabel-Operationen von v wird die Adjazenzliste von v einmal mit dem *current*-Zeiger durchlaufen, um zulässige Kanten zu suchen. Dies benötigt Zeit $\mathcal{O}(A(v))$, wenn $A(v)$ die Größe der Adjazenzliste von v ist. Jede Relabel-Operation benötigt ebenfalls Zeit $\mathcal{O}(A(v))$, da zur Bestimmung des Minimums in Zeile 14 die Adjazenzliste einmal durchlaufen wird. Relabel-Operationen von v und das Suchen von zulässigen Kanten von v aus benötigen also über die gesamte Ausführung des Algorithmus hinweg Laufzeit $\mathcal{O}(|V|A(v))$.

Damit ergibt sich für alle Knoten zusammen eine Laufzeit von $\mathcal{O}(\sum_{v \in V} |V|A(v)) = \mathcal{O}(|V||E|)$, da alle Adjazenzlisten zusammen gerade alle Kanten enthalten. □

Lemma 6.4. *Der Algorithmus führt* $\mathcal{O}(|V||E|)$ *sättigende Pushes aus.*

Beweis. Strategie: Ich will zeigen, dass zwischen je zwei sättigenden Pushes entlang einer Kante (v, w) die Höhe von w um mindestens 2 Einheiten ansteigen muss. Da die Höhe von w im Verlauf des Algorithmus monoton steigend ist und durch $\mathcal{O}(|V|)$ beschränkt ist folgt, dass entlang einer Kante maximal $\mathcal{O}(|V|)$ sättigende Pushes erfolgen können. Daraus folgt dann die Behauptung.

Erfolge also entlang einer Kante (v, w) ein sättigender Push. Zu diesem Zeitpunkt seien die jeweiligen Höhen $h(v)$ und $h(w)$. Da die Kante (v, w) vor dem Push im Rest-Netzwerk enthalten sein muss, gilt nach I3:

$$h(v) \leq h(w) + 1$$

Gleichzeitig ist (v, w) aber eine zulässige Kante, also gilt $h(w) < h(v)$. Insgesamt folgt:

$$h(v) = h(w) + 1$$

Nach dem sättigenden Push existiert die Kante (v, w) zunächst nicht mehr im Rest-Netzwerk. Damit sie wieder ins Rest-Netzwerk aufgenommen werden kann, muss von w nach v gepusht werden. Dazu muss (w, v) zulässig werden, d.h. für die Höhen $h'(v)$ und $h'(w)$ zu diesem Zeitpunkt muss gelten:

$$h'(w) > h'(v) \geq h(v) = h(w) + 1 \implies h'(w) \geq h(w) + 2$$

Danach muss mindestens v wieder relabelt werden, damit ein weiterer sättigender Push von w nach v erfolgen kann. In jedem Fall gilt aber, dass zwischen zwei sättigenden Pushes entlang der Kante (v, w) die Höhe von w um mindestens zwei Einheiten ansteigen muss. Da die Höhe von w monoton steigend und durch $2|V|$ beschränkt ist, können entlang der Kante (v, w) nur $\mathcal{O}(|V|)$ sättigende Pushes ausgeführt werden.

Diese Abschätzung gilt für jede Kante im Fluss-Netzwerk, insgesamt ist die Anzahl der sättigenden Pushes also durch $\mathcal{O}(|V||E|)$ beschränkt. □

Als nächstes will ich mit Hilfe einer Potentialfunktion eine Obergrenze für die Anzahl der nichtsättigenden Pushes zeigen.

Lemma 6.5. *Der Algorithmus führt $\mathcal{O}(|V|^2|E|)$ nichtsättigende Pushes aus.*

Beweis. Sei $I \subset V$ die Menge der aktiven Knoten. Definiere die Potentialfunktion Φ durch

$$\Phi := \sum_{v \in I} h(v)$$

Nach Aufruf von PREPROCESS gilt wegen $|I| < |V|$ und $h(v) < |V|$ für $v \in I$ für den Anfangswert $\Phi < |V|^2$. Jetzt gilt es zu untersuchen, wie sich Φ während der Ausführung des Algorithmus ändern kann. Die Knotenmenge I wird nur durch PUSH, die Höhenfunktion nur durch RELABEL geändert.

Relabel-Operation auf v: $h(v)$ und damit Φ steigt. Nach Lemma 6.1 kann dadurch während des gesamten Ablaufs des Algorithmus pro Knoten maximal eine Erhöhung um $2|V|$ erreicht werden. Insgesamt ist also die Erhöhung von Φ durch Relabel-Operationen durch $2|V|^2$ beschränkt.

Sättigender Push entlang (v, w): Dadurch wird der Knoten w möglicherweise aktiv, Φ kann also um bis zu $h(w)$ Einheiten erhöht werden. Nach Lemma 6.4 werden maximal $|V||E|$ sättigende Pushes ausgeführt, und $h(w)$ ist außerdem durch $2|V|$ beschränkt (Lemma 6.1). Insgesamt kann Φ also über die gesamte Laufzeit des Algorithmus durch sättigende Pushes um maximal $2|V|^2|E|$ erhöht werden.

Nichtsättigender Push entlang (v, w): Da aller Überschuss aus v entfernt wird, ist v nicht mehr aktiv und Φ reduziert sich um $h(v)$. Gleichzeitig kann aber w aktiv werden und Φ sich damit um $h(w)$ erhöhen. Da (v, w) eine zulässige Kante war, gilt $h(v) > h(w)$. Insgesamt verringert sich Φ also pro nichtsättigendem Push mindestens um 1.

Mit dem Anfangswert von Φ und den oberen Schranken für Relabel-Operationen und sättigende Pushes ergibt sich, dass Φ im Verlauf des Algorithmus um maximal $|V|^2 + 2|V|^2 + 2|V|^2|E|$ erhöht werden kann. Nichtsättigende Pushes reduzieren Φ jeweils mindestens um 1. Da Φ nicht negativ werden kann ergibt sich als obere Schranke für die Anzahl der nichtsättigenden Pushes $3|V|^2 + 2|V|^2|E| = \mathcal{O}(|V|^2|E|)$. □

Lemma 6.6. *Ein Aufruf von* PUSH *hat Laufzeit* $\mathcal{O}(1)$.

Beweis. PUSH führt lediglich eine konstante Zahl einfacher arithmetischer Operationen aus und aktualisiert bei Bedarf die Liste der aktiven Knoten, was wiederum nur konstante Laufzeit benötigt. □

Fasst man die Ergebnisse aus diesem Abschnitt zusammen, so sieht man, dass die Laufzeit von PREFLOW-PUSH durch nichtsättigende Pushes dominiert wird. Diese benötigen nach Lemma 6.5 und 6.6 Laufzeit $\mathcal{O}(|V|^2|E|)$. Damit ist der folgende Satz bewiesen.

Satz 6.7. *Der generische Preflow-Push Algorithmus hat eine Worst-Case-Laufzeit von* $\mathcal{O}(|V|^2|E|)$.

Ich habe bereits darauf hingewiesen, dass der generische Preflow-Push Algorithmus offen lässt, in welcher Reihenfolge aktive Knoten und zulässige Kanten ausgewählt werden. Tatsächlich ist es möglich, die Worst-Case-Laufzeit des Preflow-Push Algorithmus zu verbessern, indem eben diese Reihenfolge genauer festgelegt wird.

7 Der FIFO Preflow-Push Algorithmus

Wie bereits erwähnt wird die Laufzeit von PREFLOW-PUSH durch nichtsättigende Pushes dominiert. Immer wenn ein Knoten aktiv wird, folgt später ein von diesem Knoten ausgehender nichtsättigender Push, der den Knoten wieder inaktiv werden lässt. Beim FIFO Preflow-Push Algorithmus wird dieser nichtsättigende Push hinausgezögert, indem aktive Knoten in FIFO-Reihenfolge bearbeitet werden, und von jedem Knoten aus so lange gepusht wird, bis dieser entweder inaktiv wird oder relabelt werden muss. Der Algorithmus speichert demnach aktive Knoten in einer FIFO L, und seine Hauptroutine sieht wie folgt aus.

```
FIFO-PPA()
1  PREPROCESS()
2  while head[L] ≠ nil
3        do v ← FIFO-DEQUEUE(L)
4           while v ist aktiv und ∃ zulässige Kante (v, w)
5                 do PUSH(v, w)
6           if v ist aktiv
7              then RELABEL(v)
8                   FIFO-ENQUEUE(L, v)
```

Außerdem müssen PUSH und PREPROCESS aktiv gewordene Knoten natürlich in die FIFO einfügen.

Um die Laufzeit dieses Algorithmus zu analysieren, führe ich sogenannte *Phasen* ein. Eine Phase ist ein Zeitabschnitt in der Ausführung des Algorithmus, während dem eine bestimmte Menge von aktiven Knoten einmal durch den Schleifenrumpf von Zeile 4 bis 8 bearbeitet wird. Die Phasen sind dabei wie folgt rekursiv definiert: Während der ersten Phase werden alle Knoten bearbeitet, die direkt nach der Ausführung von PREPROCESS aktiv sind (und damit in der FIFO L stehen). Während der zweiten Phase werden alle Knoten bearbeitet, die nach Beendigung der ersten Phase aktiv sind, und allgemein werden in der $(n+1)$-ten Phase alle Knoten bearbeitet, die nach Beendigung der n-ten Phase aktiv sind.

Lemma 7.1. FIFO-PPA *führt pro Phase* $\mathcal{O}(|V|)$ *nichtsättigende Pushes aus.*

Beweis. Während einer Phase werden gerade die Knoten bearbeitet, die zu einem festen Zeitpunkt in der FIFO L stehen. Da jeder Knoten höchstens einmal in L stehen kann, wird jeder Knoten höchstens einmal pro Phase von der äußeren Schleife in FIFO-PPA untersucht. Dadurch geht aber von jedem Knoten höchstens ein nichtsättigender Push pro Phase aus (danach wird der Knoten inaktiv). Daraus folgt die Behauptung. □

Lemma 7.2. FIFO-PPA *führt* $\mathcal{O}(|V|^2)$ *Phasen aus.*

Beweis. Betrachte die Potentialfunktion

$$\Phi := \max\left(\{h(v) \mid v \text{ ist ein aktiver Knoten}\} \cup \{0\}\right)$$

und untersuche die Änderung von Φ im Verlauf einer Phase. Beachte zunächst, dass Φ durch Knoten, die im Verlauf einer Phase durch Pushes aktiv werden, nicht verändert wird. Das liegt daran, dass nur in Richtung niedrigerer Knoten gepusht wird. Unterscheide nun zwei Fälle für den Verlauf einer Phase:

1. Fall („Relabel-Phase"): Der Algorithmus relabelt während der Phase mindestens einen Knoten. Dadurch kann Φ maximal um so viele Einheiten ansteigen, wie ein Knoten gestiegen ist. Da die Höhe jedes einzelnen Knoten durch $2|V|$ beschränkt ist, ist der Anstieg von Φ über alle Relabel-Phasen durch $2|V|^2$ beschränkt.

2. Fall („Entleerungs-Phase"): Der Algorithmus relabelt während der Phase keinen Knoten. Dann entleeren alle Knoten, die zu Beginn der Phase aktiv waren, ihren Überschuss in Richtung von tiefergelegenen Knoten und werden inaktiv. Deshalb nimmt Φ in jeder Entleerungs-Phase um mindestens eine Einheit ab.

Zu Beginn der Ausführung ist $\Phi = 0$, da PREPROCESS die Höhen aller Knoten auf 0 initialisiert (mit Ausnahme der Quelle, die nach Definition nie aktiv ist). Da Φ maximal um $2|V|^2$ Einheiten ansteigt und nie negativ werden kann, ist die Anzahl der Entleerungs-Phasen durch $2|V|^2$ beschränkt. Da der Algorithmus höchstens $2|V|^2$ Relabel-Operationen ausführt ist auch die Anzahl der Relabel-Phasen durch $2|V|^2$ beschränkt. Insgesamt ist die Anzahl der Phasen also $\mathcal{O}(|V|^2)$. □

Satz 7.3. *Die Laufzeit von* FIFO-PPA *ist* $\mathcal{O}(|V|^3)$.

Beweis. FIFO-PPA unterscheidet sich vom generischen Preflow-Push Algorithmus nur dadurch, dass die Reihenfolge der Operationen genauer festgelegt wird und die aktiven Knoten in einer FIFO gespeichert werden. Auch FIFO-Operationen benötigen konstante Laufzeit. Die Abschätzungen aus dem letzten Abschnitt gelten also auch für FIFO-PPA.

Aus Lemma 7.1 und 7.2 folgt aber, dass nur Zeit $\mathcal{O}(|V|^3)$ für nichtsättigende Pushes aufgewendet wird. Sättigende Pushes, Relabel-Operationen und das Suchen von zulässigen Kanten benötigen nach wie vor Zeit $\mathcal{O}(|V||E|)$. Für alle Graphen gilt $|E| \leq |V|^2$, also folgt die Behauptung. □

Diese Laufzeit ist tatsächlich besser als $\mathcal{O}(|V|^2|E|)$, da im schlimmsten Fall $|E| = \mathcal{O}(|V|^2)$ und damit $\mathcal{O}(|V|^2|E|) = \mathcal{O}(|V|^4)$ gilt.

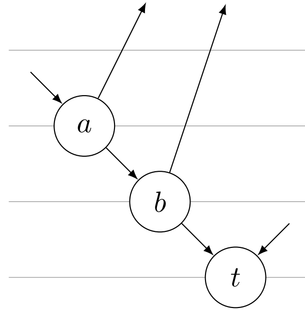


Abbildung 6: Wenn a und b gleichzeitig aktiv sind sollte a zuerst bearbeitet werden.

8 Der Highest-Label Preflow-Push Algorithmus

Während der Ausführung des Preflow-Push Algorithmus wird die Höhe eines Knotens um so größer, je weiter dieser Knoten von der Senke entfernt ist. In Abb. 6 muss Überschuss vom höheren Knoten a zuerst über den Knoten b fließen, der näher an der Senke liegt. Wenn a und b gleichzeitig aktiv sind, sollte zuerst Knoten a entleert werden, damit danach aller angesammelter Überschuss von b auf einmal in die Senke geschickt werden kann (natürlich unter der Voraussetzung, dass die nötigen Restkapazitäten vorhanden sind). Bei einer umgekehrten Bearbeitungsreihenfolge würde zunächst ein nichtsättigender Push von b nach t stattfinden, der b inaktiv werden lässt. Aber gleich im Anschluss wird b durch den Push von a wieder aktiv und letztendlich muss ein zweiter nichtsättigender Push entlang der Kante (b, t) durchgeführt werden.

Der Highest-Label Preflow-Push Algorithmus bearbeitet deshalb immer die höchsten aktiven Knoten zuerst und vermeidet damit unnötige nichtsättigende Pushes.

Satz 8.1. *Der Highest-Label Preflow-Push Algorithmus hat Laufzeit* $\mathcal{O}(|V|^2\sqrt{|E|})$.

Einen Beweis dieses Satzes findet man in Kapitel 7.8 von [Ahu]. Diese Laufzeit ist besser als $\mathcal{O}(|V|^3)$, da für alle Graphen $|E| \leq |V|^2$ gilt.

9 Zusammenfassung und Ausblick

Ich habe Fluss-Netzwerke und das Maximum-Flow Problem zur direkten Modellierung von real auftretenden Fragestellung vorgestellt und gezeigt, dass Preflow-Push Algorithmen das Maximum-Flow Problem effizient lösen. Die Lösung in polynomieller Zeit wird ganz wesentlich dadurch ermöglicht, dass ungeschickte Entscheidungen des Algorithmus auch noch spät während der Ausführung mit Hilfe des Rest-Netzwerks schnell und elegant rückgängig gemacht werden können. Dadurch wird ein Backtracking-Verfahren, bei dem man womöglich einen Suchbaum exponentieller Größe durchlaufen müsste, unnötig.

Weitere, zum Teil heuristische Möglichkeiten zur Verbesserung der Laufzeit werden in Kapitel 7 von [Ahu] genannt. So können die Höhen der Knoten durch eine Breitensuche mit dem Abstand der Knoten von der Senke initialisiert werden. Dadurch werden Pushes schneller in eine richtige Richtung geleitet, da ein guter Weg durch die Höhenunterschiede der Knoten bereits vorgezeichnet ist. Eine weitere Verbesserung ergibt sich

aus der Beobachtung, dass Preflow-Push Algorithmen oft schnell den maximal erreichbaren Überschuss in der Senke finden, dann aber noch sehr viel Zeit benötigen, um den Überschuss der anderen Knoten in die Quelle zurückfließen zu lassen. Eine Heuristik kann versuchen, diese Situation zu erkennen um dann alle Knoten so weit anzuheben, dass verbleibende Überschüsse schneller in die Quelle zurückfinden.

Neben den sehr anschaulichen Problemen, auf die die Einführung hingedeutet hat, gibt es eine Vielzahl weiterer kombinatorischer Probleme, die auf das Maximum-Flow Problem reduziert werden können, wie zum Beispiel das Finden von Matchings in bipartiten Graphen (vgl. Kapitel 26 von [CLRS] und Kapitel 6 von [Ahu]).

Literatur

[Ahu] Ahuja, Ravindra K., Magnanti, Thomas L., Orlin, James B.: *Network Flows: theory, algorithms and applications*, Prentice-Hall, Inc. (1993)

[CLRS] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford: *Introduction to Algorithms, Second Edition*, MIT Press (2001)