Neue Algorithmen für das Maximum-Flow-Problem

Torben Hagerup

A 90/08

Mai 1990

FB 14, Informatik
Universität des Saarlandes
D–6600 Saarbrücken
West Germany

This research was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

Neue Algorithmen für das Maximum-Flow-Problem

TORBEN HAGERUP*

Fachbereich Informatik
Universität des Saarlandes
D–6600 Saarbrücken

Wir betrachten das folgende Problem: Gegeben ist ein *Netzwerk*, d.h. ein gerichteter Graph G, eine positive ganzzahlige *Kapazität* für jede Kante in G und zwei ausgezeichnete Knoten in G, die *Quelle* s und die *Senke* t. Das Ziel ist, möglichst viel Fluß durch das Netzwerk von s nach t zu transportieren, wobei der Fluß über eine Kante die Kapazität der Kante nicht überschreiten kann und an jedem Knoten außer s und t *Flußerhaltung* herrschen muß ("Fluß hinein = Fluß heraus"). Konkret stelle man sich s als ein Ölfeld, t als eine Großstadt und das Netzwerk als ein System von Rohrleitungen vor, das die Stadt mit Öl versorgen soll. Wir müssen den Ölfluß über jede Rohrleitung so regulieren, daß die Ölmenge, die die Stadt insgesamt erreicht, maximiert wird.

Das Maximum-Flow-Problem hat eine lange Geschichte und erhebliche praktische Bedeutung. Die Vorgehensweise der meisten Algorithmen für das Problem besteht darin, "Flußeinheiten" durch das Netz hin und her zu schieben, bis ein maximaler Fluß erreicht ist. Sei n die Knotenanzahl, m die Kantenanzahl und U die größte Kapazität einer Kante. Ahuja und Orlin haben einen einfachen Algorithmus angegeben, der einen maximalen Fluß unter Ausführung von $O(nm+n^2 \log U)$ elementaren Flußoperationen berechnet. Wir beschreiben diesen Algorithmus und skizzieren Verbesserungen, die es erlauben, die Anzahl der Flußoperationen auf $O(n^{3/2}m^{1/2} + n^2 \log U)$ zu reduzieren (gemeinsame Arbeit mit Joseph Cheriyan und Kurt Mehlhorn). Für $m = \Theta(n^2)$ und U nicht außerordentlich groß ist die bewiesene obere Schranke für die Anzahl der Flußoperationen $O(n^{2.5})$, gegenüber $O(n^3)$ für den besten bisher bekannten Algorithmus.

1. Netzwerke und ihre Anwendungen

Ein (gerichteter) *Graph* ist mathematisch gesehen ein Paar (V, E), wobei V eine endliche Menge und E eine Teilmenge von $V \times V$ ist ($V \times V$ ist die Menge aller geordneter Paare der Form (v, w), wobei $v, w \in V$). Die Elemente aus V sind die *Knoten*, die Elemente aus E die *Kanten* des Graphen. Graphen werden üblicherweise so dargestellt, daß ein Knoten als ein Punkt oder ein Kreis in der Ebene und eine Kante (v, w) als ein Pfeil von der Repräsentation von v zur Repräsentation von w gezeichnet wird. Ein Beispiel ist in Abb. 1 zu sehen.

* Partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

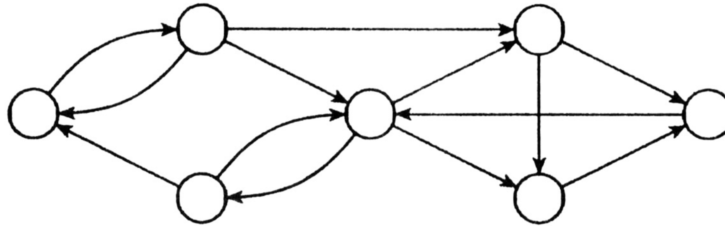

Abb. 1. Ein gerichteter Graph.

Graphen haben sehr viele Interpretationen und Anwendungen. Eine Kante (v, w) könnte z.B. bedeuten, daß Firma v Kunde der Firma w ist, daß Spieler v den Spieler w besiegt hat, oder daß Arbeitsgang v (z.B. Fundament gießen) ausgeführt sein muß, bevor Arbeitsgang w (z.B. Fenster einsetzen) anfangen kann. Wir gehen darauf nicht näher ein. Wird jeder Kante eines Graphen ein *Wert* aus $\mathbb{N} = \{0, 1, 2, \dots\}$ zugeordnet, spricht man von einem *Netzwerk* (normalerweise werden beliebige reelle Kantenwerte zugelassen, aber wir beschränken uns hier auf nicht-negative ganze Kantenwerte). Abb. 2 zeigt ein Beispielnetzwerk. Formal definieren wir ein Netzwerk als ein Tripel (V, E, g), wobei (V, E) ein gerichteter Graph und $g : E \to \mathbb{N}$ eine Funktion ist.

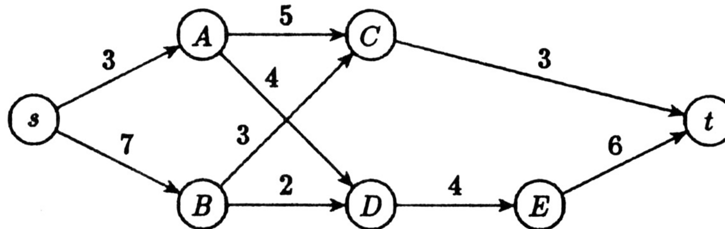

Abb. 2. Ein Netzwerk. Jede Kante ist mit ihrem Wert beschriftet.

Die vielleicht bekanntesten Beispiele für Netzwerke sind Verkehrsnetze aller Art. Die Knoten sind dabei die Punkte, zwischen denen der Verkehr verläuft (Städte, Straßenkreuzungen, Häfen, Bahnhöfe, Flughäfen), und eine Kante (v, w) stellt den (direkten) Weg von v nach w dar. Der Wert einer Kante repräsentiert dabei, je nach Bedarf, die *Länge* des entsprechenden Weges, die *Kosten* oder die *Verzögerung* (Reisezeit), die mit seiner Benutzung verbunden sind, oder wiederum ganz andere Parameter. Weitere Beispiele für Netzwerke sind elektrische Schaltkreise, wo Kantenwerte z.B. Widerstände repräsentieren können. Dient ein Netzwerk zur Beförderung von irgendeiner "Ware", wie in dem eingangs erwähnten Ölbeispiel oder im Falle von Telefonleitungen, ist es oft sinnvoll, durch einen Kantenwert eine *Kapazität*, d.h. ein maximales Beförderungsvermögen, auszudrücken. Diese Betrachtungsweise kann auch bei Verkehrsnetzen angelegt werden, wobei eine globalere Sicht zum Ausdruck gebracht wird. Während ein individueller Autofahrer meist bemüht ist, seine Reisezeit zu minimieren, interessiert sich ein Verkehrsplaner eher für die Kapazität einer existierenden oder geplanten Straße, oder vielleicht eher noch für den genauen Zusammenhang zwischen Verkehrsaufkommen und Reisezeit. Kapazitäten werden in Mengeneinheiten pro Zeiteinheit gemessen, also z.B. in hl/Sek. oder in Autos/Min.

Es gibt weitere Anwendungen von Netzwerken, wo das Netzwerk nicht physikalisch vorhanden ist, sondern nur bestimmte Relationen zwischen Objekten ausdrückt. Eine Zentralstelle für die Vergabe von Studienplätzen könnte z.B. zu Entscheidungszwecken ein Netzwerk aufbauen, dessen Knoten teils Studenten, teils Studienplätze repräsentieren. Eine Kante zwischen einem Studenten und einem Studienplatz gibt an, daß der Studienplatz überhaupt für den betreffenden Studenten in Frage kommt, und ihr Wert ist ein Maß dafür, wie "wünschenswert" die entsprechende Vergabe ist. Das Auffinden einer optimalen Gesamtvergabe, wie auch immer definiert, kann jetzt als

Netzwerkproblem studiert und gelöst werden. Zusammenfassend läßt sich sagen, daß Netzwerke eine bedeutende Rolle in Produktionsplanung, VLSI-Entwurf, Verkehrsplanung und Telekommunikation sowie auf vielen anderen Gebieten spielen.

Eine fundamentale Frage, die man bei einem vorliegenden Netzwerk mit zwei Knoten s und t stellen kann, lautet: Wenn Kantenwerte als Weglängen (oder Kosten, oder Verzögerungen) interpretiert werden, welches ist dann der kürzeste (billigste, schnellste) Weg von s nach t? Dies ist das intensiv untersuchte Problem der kürzesten Wege (s. z.B. Tarjan, 1983, Kap. 7). Eine fast ebenso fundamentale Frage ist die folgende: Wenn Kantenwerte als Kapazitäten interpretiert werden, wie kann man die größtmögliche Flußmenge ("Warenmenge") von s nach t schicken? Dies ist das Maximum-Flow-Problem, mit dem wir uns hier befassen werden. Abb. 3 zeigt als Beispiel einen maximalen Fluß in dem Netzwerk aus Abb. 2.

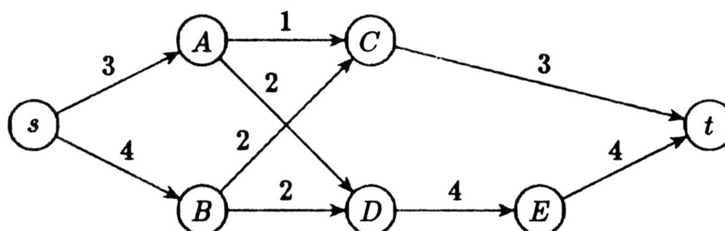

Abb. 3. Ein maximaler Fluß in dem Netzwerk aus Abb. 2. Jede Kante ist mit der über sie verlaufenden Flußmenge beschriftet.

Ein *Weg* der *Länge* k von v nach w in einem Graphen (V, E) oder in einem Netzwerk (V, E, g) ist eine Knotenfolge $v_0, v_1, \ldots, v_k$, wobei $v_0 = v$, $v_k = w$ und $(v_{i-1}, v_i) \in E$, für $i = 1, \ldots, k$. Wir benutzen im folgenden den Begriff "Länge eines Weges" immer in dieser Bedeutung, nicht in der Bedeutung einer konkreten Interpretation (wie z.B. Summe der Kantenwerte auf dem Weg).

2. Das Maximum-Flow-Problem

Gegeben sei ein Netzwerk $G = (V, E, cap)$ und zwei Knoten $s, t \in V$. $cap(v, w)$ wird als die *Kapazität* der Kante (v, w) bezeichnet, s ist die *Quelle* und t ist die *Senke*. Wir nehmen an, daß G *symmetrisch* ist, d.h., daß $(v, w) \in E \Rightarrow (w, v) \in E$, für alle $v, w \in V$, was durch Einführung von zusätzlichen Kanten mit Kapazität 0 leicht erreicht werden kann. Formal definieren wir einen *Fluß* in G als eine Funktion $f : E \to \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. $f(v, w)$ ist dabei der Nettofluß von v nach w über die Kanten (v, w) und (w, v). Trägt z.B. die Kante (v, w) den Fluß 5 und (w, v) den Fluß 2, setzen wir $f(v, w) = 3$ und $f(w, v) = -3$. Ein Fluß f muß folgende Bedingungen erfüllen:

(1) $f(v, w) = -f(w, v)$, für alle $(v, w) \in E$ (Antisymmetrie);
(2) $f(v, w) \le cap(v, w)$, für alle $(v, w) \in E$ (Kapazitätsbeschränkung);
(3) $\sum_{u:(u,v) \in E} f(u, v) = 0$, für alle $v \in V \setminus \{s, t\}$ (Flußerhaltung).

Bedingung (1) ist, wie oben gesehen, eine reine Konvention ohne physikalischen Inhalt. Bedingung (2) besagt, daß der Fluß über eine Kante deren Kapazität nicht überschreiten kann. Bedingung (3), schließlich, summiert für jeden Knoten $v \in V \setminus \{s, t\}$ die Nettoflüsse über Kanten, die in v münden, und fordert, daß diese Summe Null ist. Anders gesagt, der gesamte Fluß, der v verläßt, muß genau so groß sein wie der gesamte Fluß, der in v hineintritt. Das entspricht unserer Annahme, daß Fluß von s nach t geschickt wird, daß Fluß also nur an s "produziert" und nur an t "verbraucht" wird.

Der *Wert* von f ist $\sum_{v:(v,t)\in E} f(v,t)$, der Nettofluß, der t erreicht. Wegen der Flußerhaltung an allen Knoten außer s und t ist das das gleiche wie $\sum_{v:(s,v)\in E} f(s,v)$, der Nettofluß, der s verläßt. Ein ***maximaler Fluß*** ist ein Fluß, der unter allen Flüssen den größtmöglichen Wert hat. Wir suchen einen effizienten Algorithmus, der einen maximalen Fluß berechnet.

Die ***Restkapazität*** einer Kante (v,w) wird als $rescap(v,w) = cap(v,w) - f(v,w)$ definiert. Die Kante heißt ***saturiert***, falls $rescap(v,w) = 0$; sonst ist sie eine ***Restkante***. E_{res} bezeichnet die Menge aller (augenblicklichen) Restkanten, und G_{res} bezeichnet den ***Restgraphen*** (V, E_{res}). Sei im folgenden $n = |V|$, $m = |E|$ und $U = \max(\{1\} \cup \{cap(v,w) : (v,w) \in E\})$.

3. Der generische Algorithmus

Dieser Abschnitt beschreibt den sogenannten generischen Maximum-Flow-Algorithmus, der in (Goldberg and Tarjan, 1988) angegeben wurde.

Betrachten wir das Netzwerk aus Abb. 2. Versucht man, "von Hand" einen maximalen Fluß in einem so kleinen Netzwerk zu bestimmen, scheint es natürlich, zunächst so viele "Flußeinheiten" von s über die Kanten der Form (s,v) zu schicken, wie diese zulassen, um danach zu versuchen, diese Flußeinheiten zu t weiterzuleiten. Wendet man dieses Verfahren auf das Netzwerk aus Abb. 2 an, sieht man leicht, daß die 3 Flußeinheiten, die von s zu A geschickt werden, ohne weiteres über den Weg A, C, t die Senke erreichen können. Von den 7 Einheiten, die zu B geschickt werden, können 2 den Weg B, D, E, t nehmen, weitere 2 können über den Weg B, C, A, D, E, t zu t gelangen (Fluß über die Kante (C,A) schicken heißt in Wirklichkeit, den Fluß über (A,C) verringern), und die restlichen 3 Einheiten müssen zu s zurückgeschickt werden, wodurch man den maximalen Fluß in Abb. 3 erhält.

Der Algorithmus geht tatsächlich ähnlich vor. Zunächst brauchen wir Bezeichnungen für die auftretenden Konzepte.

Für jeden Knoten $v \in V$ sei $e(v) = \sum_{u:(u,v)\in E} f(u,v)$. Aus naheliegenden Gründen nennen wir $e(v)$ den ***Überschuß*** an v. Falls f ein Fluß ist, dann ist $e(v) = 0$ für alle $v \in V\backslash\{s,t\}$ (Bedingung (3)). Im obigen Beispiel gab es aber an manchen Knoten positiven Überschuß. Um diese Situation zu beschreiben führen wir den Begriff des ***Präflusses*** ein. Ein Präfluß ist eine Funktion $f : E \to \mathbf{Z}$, die Bedingungen (1) und (2) oben sowie folgende abgeschwächte Form der Bedingung (3) erfüllt:

(3') $e(v) \geq 0$, für alle $v \in V\backslash\{s,t\}$.

Die Grundoperation, mit der wir im Beispiel den aktuellen Präfluß verändert haben, läßt sich ohne Schwierigkeiten als die Bewegung von c Flußeinheiten über eine Kante (v,w) von v nach w identifizieren, wobei c eine positive ganze Zahl ist. Formal erhöhen wir einfach $f(v,w)$ um den Wert c (und verringern $f(w,v)$ um c). Wir nennen diese Operation einen ***Push*** der Größe c über die Kante (v,w). Da wir es weiterhin mit einem Präfluß zu tun haben wollen, fordern wir einerseits, daß $c \leq rescap(v,w)$, so daß die Kapazitätsbedingung für die Kante (v,w) nicht verletzt wird, und andererseits, daß $c \leq e(v)$, so daß der Überschuß an v nicht-negativ bleibt. Der Push heißt ***saturierend***, wenn $c = rescap(v,w)$.

Es reicht natürlich nicht, Fluß mit Hilfe von Pushes planlos durch das Netzwerk hin und her zu schicken; der Fluß sollte sich im großen und ganzen in Richtung von s nach t bewegen. Dies zu erreichen ist ein weitaus schwierigeres Problem, als es auf den ersten Blick erscheinen mag, und es ist bis heute kein Algorithmus bekannt, der es vermeidet, über die meisten Kanten Fluß wiederholt hin und her zu schicken. Am Anfang scheint es zwar sinnvoll, Fluß entlang kürzester

Wege nach t zu schicken, aber sobald die kürzesten Wege alle saturiert sind, muß man anders vorgehen (s. Abb. 3 als Beispiel dafür, daß maximale Flüsse im allgemeinen auch nicht-kürzeste Wege benutzen müssen). Es scheint angebracht, die kürzesten Wege nicht in dem ursprünglichen Netzwerk, sondern im (aktuellen) Restgraphen zu suchen, indem man also nur die Kanten in Betracht zieht, die noch zusätzlichen Fluß aufnehmen können. Anders formuliert: Für alle $v \in V$, sei $\delta(v)$ die Länge eines kürzesten Weges in G_{res} von v nach t ($\delta(v) = \infty$ falls kein solcher Weg existiert). Wollen wir überschüssigen Fluß an einem Knoten v loswerden, schicken wir ihn über eine Kante (v, w), für die $\delta(w) = \delta(v) - 1$ (eine solche Kante existiert zwangsläufig, sofern $v \neq t$ und $\delta(v) \neq \infty$). Intuitiv ist der Fluß damit näher zu t gekommen, und wir haben Fortschritt erzielt.

Der Algorithmus berechnet nicht wirklich die Funktion δ, sondert verwaltet auf ausgeklügelte Weise eine (grobe) Annäherung d zu δ, die ebenso gute Dienste leistet. Wir definieren eine *Beschriftung* als eine Funktion $d : V \to \mathbb{N}$. Die Beschriftung heißt ***legal***, falls

$$d(w) \geq d(v) - 1, \qquad \text{für alle } (v, w) \in E_{res}.$$

Wir benutzen jetzt an Stelle von δ eine beliebige legale Beschriftung d. Es ist sehr nützlich, sich $d(v)$ als die "Höhe" von v über einer Normalebene vorzustellen. Die Bedingung der Legalität sagt dann gerade, daß Restkanten zwar "abfallen", aber nicht "steil abfallen" dürfen (der Endknoten einer Restkante darf nur um eine Einheit tiefer liegen als der Startknoten). Außerdem wollen wir, wie schon oben angedeutet, Fluß nur über Restkanten schicken, die abfallen. Solche Kanten nennen wir ***wählbar***.

Die Grundidee ist jetzt, s auf einen hohen Gipfel zu setzen, t in der tiefsten Ebene festzuhalten und darauf zu warten, daß der gesamte Fluß in Folge von Pushes von s nach t "hinunterströmt". Ganz so einfach geht es allerdings nicht, denn wir brauchen zusätzlich zu Pushes eine zweite Operation, ***relabel***, die uns erlaubt, unsere "Landschaft" zu verändern: Kann der Überschuß einen Knoten v nicht verlassen, weil alle Restkanten der Form (v, w) geradeaus führen oder ansteigen, heben wir v an, indem wir $d(v)$ um 1 erhöhen (vgl. die übliche Methode, Zeltdecken von Wasser zu befreien). Dadurch entstehen hoffentlich Restkanten der Form (v, w), die abfallen, aber natürlich keine solchen, die steil abfallen. Die neue Beschriftung ist somit weiterhin legal.

Wir können unsere zwei Grundoperationen jetzt formal definieren. Wir nennen einen Knoten v ***aktiv***, falls $v \in V \setminus \{s, t\}$ und $e(v) > 0$ (ein aktiver Knoten ist ein Knoten, an dem "noch etwas zu tun ist").

push(v, w, c).
Vorbedingung: v ist aktiv, (v, w) ist wählbar, $c \in \mathbb{N}$ und $1 \leq c \leq \min\{rescap(v, w), e(v)\}$.
Setzt $f(v, w) := f(v, w) + c$ und $f(w, v) := f(w, v) - c$.

relabel(v).
Vorbedingung: v ist aktiv, und es gibt keine wählbare Kante der Form (v, w).
Setzt $d(v) := d(v) + 1$.

Eine Push- oder Relabel-Operation mit erfüllter Vorbedingung nennen wir ***erlaubt***. Der gesamte generische Algorithmus, der unten wiedergegeben ist, besteht aus einem Initialisierungsteil, in dem s auf einen Gipfel gesetzt wird, die anderen Knoten vorerst in der tiefsten Ebene gelassen werden, und die Kanten der Form (s, v) saturiert werden (das ist notwendig, um die Legalität der anfänglichen Beschriftung zu gewährleisten), gefolgt von einer Schleife, in der solange erlaubte Push- oder Relabel-Operationen ausgeführt werden, bis keine Operation mehr erlaubt ist.

```
procedure Initialisiere;
  d(s) := n; for all v ∈ V\{s} do d(v) := 0;
  for all (v,w) ∈ E do f(v,w) := 0;
  for all (s,v) ∈ E do push(s,v,cap(s,v));
Generischer Maximum-Flow-Algorithmus:
  Initialisiere;
  while mindestens ein Knoten ist aktiv
  do begin
        Sei v ein aktiver Knoten;
        if mindestens eine Kante der Form (v,w) ist wählbar
        then führe eine erlaubte Push-Operation der Form push(v,w,c) aus
        else relabel(v);
     end.
```

Es ist nicht offensichtlich, daß der Algorithmus terminiert, daß es also irgendwann keine aktiven Knoten mehr gibt. Unter der Annahme, daß der Algorithmus terminiert, können wir aber leicht seine Korrektheit nachweisen (partielle Korrektheit). Zuerst zeigen wir, daß d stets legal und f stets ein Präfluß ist.

Lemma 1: Nach der Initialisierung sowie nach jeder Ausführung von einer Push- oder Relabel-Operation im Algorithmus ist d eine legale Beschriftung und f ein Präfluß.

Beweis: Man sieht leicht, daß die Bedingung nach der Initialisierung erfüllt ist. Wir haben uns auch schon davon überzeugt, daß die Ausführung einer erlaubten Relabel-Operation die Legalität der Beschriftung nicht zerstört, und daß ein erlaubter Push einen Präfluß wieder in einen Präfluß überführt. Es bleibt nur zu zeigen, daß ein erlaubter Push über eine Kante (v,w) auch die Legalität der Beschriftung erhält. Dazu müssen wir die Änderungen in G_{res} betrachten, die von dem Push hervorgerufen werden. Zwei solche Änderungen sind möglich:

(1) Die Kante (v,w) kann aus G_{res} verschwinden (wenn der Push über (v,w) saturierend ist).

(2) Die Kante (w,v) kann in G_{res} hinzukommen (wenn sie vor dem Push über (v,w) saturiert war).

Fall (1) verursacht keine Probleme: Verschwindet eine Kante aus G_{res}, gibt es entsprechend eine Bedingung weniger, die die Beschriftung erfüllen muß. In Fall (2) müssen wir zeigen, daß die Kante (w,v) nicht steil abfällt. Aber sie steigt sogar an, denn über die umgekehrte Kante (v,w) erfolgt ja ein Push. ∎

Lemma 2: Falls der Algorithmus terminiert, ist f am Ende der Berechnung ein maximaler Fluß in G.

Beweis: Betrachten wir die Situation am Ende der Berechnung. Da es keine aktiven Knoten gibt, ist f nach Definition nicht nur ein Präfluß, sondern ein Fluß. Wir weisen jetzt die Maximalität von f nach. Sei dazu S die Menge der Knoten $v \in V$, für die es einen Weg von s nach v in G_{res} gibt (am Ende der Berechnung). Für jedes $v \in S$ gibt es sogar einen solchen Weg der Länge $\leq n-1$, denn auf einem kürzesten Weg wiederholt sich kein Knoten. Da aber $d(s) = n$ und es keine starken Gefälle in G_{res} gibt, können wir daraus schließen, daß $d(v) \geq 1$ für alle $v \in S$. Insbesondere ist $t \notin S$, denn $d(t) = 0$. Wir haben also eine (nur) produzierende "Region" S und eine (nur) verbrauchende "Region" $V \setminus S$ (s. Abb. 4), und jede Kante (v,w), die von S nach $V \setminus S$ führt, also mit $v \in S$ und $w \notin S$, ist saturiert, denn sonst wäre sie ja in G_{res}, und $v \in S$ würde $w \in S$ implizieren. Die Flußmenge, die insgesamt die "Grenze" von S nach $V \setminus S$ überschreitet,

kann also unmöglich vergrößert werden. Da f ein Fluß ist, erreicht diese ganze Flußmenge auch tatsächlich die Senke, und man sieht leicht, daß f maximal ist. ∎

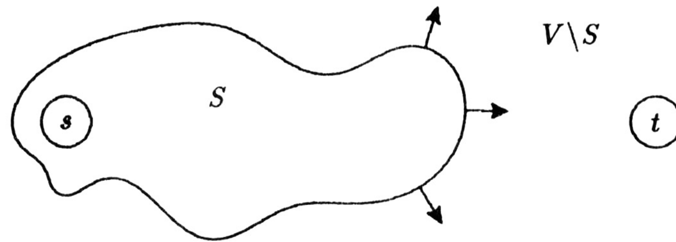

Abb. 4. Die Situation aus dem Beweis von Lemma 2. Jede Kante, die S verläßt, ist saturiert.

Nachdem die partielle Korrektheit des generischen Algorithmus feststeht, möchten wir Aussagen über seine (Terminierung und) Laufzeit machen. Dazu wollen wir die Anzahl der ausgeführten Push- und Relabel-Operationen abschätzen. Zunächst brauchen wir eine obere Schranke für die "Höhe der höchsten Gipfel".

Lemma 3: Für jeden Knoten $v \in V$ ist stets $d(v) \leq 2n - 1$.

Beweis (Skizze): Mit einem Argument wie im vorigen Beweis kann man zeigen, daß es von jedem aktiven Knoten v einen Weg zu s in G_{res} gibt (intuitiv: der Überschuß an v kann den Weg zurückverfolgen, auf dem er gekommen ist). Da ein kürzester solcher Weg Länge $\leq n - 1$ hat, $d(s) = n$ und G_{res} keine starken Gefälle hat, können wir schließen, daß $d(v) \leq 2n - 1$ für jeden aktiven Knoten v. Daraus folgt das Lemma, denn $d(v)$ wird nur dann verändert, wenn v aktiv ist. ∎

Lemma 4: Für jeden Knoten $v \in V$ wird *relabel*(v) höchstens $2n - 1$ mal ausgeführt. Die Gesamtzahl der Relabel-Operationen ist also durch $2n^2$ beschränkt.

Beweis: Folgt unmittelbar aus Lemma 3. ∎

Damit haben wir schon eine gute obere Schranke für die Anzahl der Relabel-Operationen. Wir zählen jetzt getrennt saturierende Pushes und nicht-saturierende Pushes.

Lemma 5: Es gibt höchstens n saturierende Pushes über eine feste Kante $(v, w) \in E$. Die Gesamtzahl der saturierenden Pushes is also höchstens nm.

Beweis: Sei $(v, w) \in E$. Zwischen zwei saturierenden Pushes über (v, w) muß offensichtlich ein Push über (w, v) erfolgen. Aber unmittelbar nach jedem Push über (v, w) steigt (w, v) an, während ein Push über (w, v) voraussetzt, daß (w, v) abfällt. Zwischen zwei saturierenden Pushes über (v, w) wird also $d(w)$ um mindestens 2 erhöht. Nach Lemma 4 passiert das aber höchstens $n - 1$ mal, woraus folgt, daß es höchstens n saturierende Pushes über (v, w) gibt. ∎

Die Anzahl der nicht-saturierenden Pushes ist wesentlich schwieriger in den Griff zu bekommen. Wir benutzen hierzu ein *Potentialargument*. Dabei werden Flußeinheiten mit gedachter Masse ausgestattet, so daß eine Flußeinheit am Knoten v eine potentielle Energie proportional zu $d(v)$ besitzt. Da Push-Operationen immer Flußeinheiten bergab bewegen, also mit einem Verlust an potentieller Energie verbunden sind, während die insgesamt vorhandene potentielle Energie endlich ist, können wir auf diese Weise hoffen, obere Schranken für die Anzahl der Pushes zu beweisen.

Genauer definieren wir die Potentialfunktion $\Phi = \sum_{v \in V \setminus \{s\}} e(v) \cdot d(v)$. Jeder Push bewegt mindestens eine Flußeinheit um eine Höheneinheit nach unten, verringert also Φ um mindestens 1.

Die Anzahl der Flußeinheiten, die insgesamt vorhanden sind, ist durch nU beschränkt (genauer: durch die Summe der Kapazitäten der Kanten der Form (s, v)), und diese Flußeinheiten können nach Lemma 3 höchstens die potentielle Energie $2n^2U$ haben. Würden wir nach der Initialisierung nur Push-Operationen ausführen, hätten wir damit ihre Anzahl durch $2n^2U$ beschränkt. Es gibt aber auch Relabel-Operationen, die Φ erhöhen, indem sie sozusagen "Arbeit von außen leisten". Schlimmstenfalls befinden sich alle nU Flußeinheiten immer an dem Knoten, dessen d-Wert gerade erhöht wird. Die nach Lemma 4 höchstens $2n^2$ Relabel-Operationen können Φ also höchstens um $2n^3U$ erhöhen. Damit haben wir gezeigt, daß der generische Algorithmus höchstens $2n^3U + 2n^2U$ Push-Operationen ausführt (da $\Phi = 0$ nach der Initialisierung, ist der zweite Term in Wirklichkeit überflüssig). Obwohl diese Schranke die Terminierung des Algorithmus impliziert, ist sie sehr schlecht. Im folgenden Abschnitt zeigen wir eine viel bessere Schranke für einen Spezialfall des generischen Algorithmus, bei dem die auszuführenden Operationen sorgfältiger ausgewählt werden.

4. Der skalierende Algorithmus

Die schlechte Laufzeit des generischen Algorithmus ist im wesentlichen auf zwei Faktoren zurückzuführen: (1) Wir erlauben fast wirkungslose Pushes der Größe 1, obwohl viel größere Überschüsse zur Verfügung stehen könnten; (2) Es kann sich sehr viel Fluß an einem Knoten ansammeln, so daß Relabel-Operationen beträchtliche äußere Arbeit leisten können. Wir beschreiben jetzt den in (Ahuja and Orlin, 1987) vorgestellten ***skalierenden Algorithmus***, der diese Schwierigkeiten vermeidet.

Wir führen eine ganzzahlige Variable Δ ein und interessieren uns von jetzt ab nur für Überschüsse der Größe mindestens Δ. Δ ist am Anfang ungefähr U und wird allmählich kleiner. Intuitiv leisten wir zuerst die Grobarbeit und machen uns erst danach an die Feinheiten. Um die Ansammlung von großen Flußmengen an einem Knoten zu vermeiden schicken wir erstens höchstens Δ Flußeinheiten auf einmal, zweitens schicken wir diese von einem Knoten, der unter den Knoten mit Überschuß $\geq \Delta$ minimale Höhe hat. Da der Fluß abwärts fließt, kann er damit auf keinen Knoten mit Überschuß $\geq \Delta$ treffen, so daß nach dem Push kein Knoten Überschuß $\geq 2\Delta$ hat, wenn das nicht schon vor dem Push der Fall war. Es folgt der vollständige Algorithmus.

Der skalierende Algorithmus:

Initialisiere;
$\Delta := 2^{\lfloor \log_2 U \rfloor}$;
while mindestens ein Knoten ist aktiv
do begin
  while $\max\{e(v) : v$ ist aktiv$\} < \Delta$
  do $\Delta := \Delta/2$;
  Sei v ein aktiver Knoten mit $e(v) \geq \Delta$ und minimalem d-Wert
    unter allen aktiven Knoten mit Überschuß $\geq \Delta$;
  if mindestens eine Kante der Form (v, w) ist wählbar
  then führe eine erlaubte Push-Operation der Form *push*$(v, w, \min\{\Delta, \textit{rescap}(v, w)\})$ aus
  else *relabel*(v);
end.

Eine Zeitspanne zwischen zwei aufeinanderfolgenden Änderungen von Δ nennen wir eine *Phase*.

Lemma 6: Nach der Initialisierung und nach jedem Push im skalierenden Algorithmus ist $e(v) < 2\Delta$ für jeden Knoten $v \in V \setminus \{s, t\}$.

Beweis: Vor der ersten Phase ist $e(v) \leq U < 2 \cdot 2^{\lfloor \log_2 U \rfloor}$. Am Anfang von jeder folgenden Phase ist $e(v) < 2\Delta$, denn am Ende der vorherigen Phase, also bevor Δ halbiert wurde, war $e(v) < \Delta$ (sonst wäre die Phase nicht zu Ende gegangen). Wir haben schon oben gesehen, daß die Bedingung während einer Phase nicht verletzt wird, womit das Lemma bewiesen ist. ∎

Lemma 7: Die Anzahl der vom skalierenden Algorithmus ausgeführten nicht-saturierenden Pushes ist höchstens $8n^2(\log_2 U + 1)$.

Beweis: Da es höchstens $\lfloor \log_2 U \rfloor + 1$ Phasen gibt (Δ wird nie kleiner als 1), genügt es zu zeigen, daß innerhalb jeder Phase höchstens $8n^2$ nicht-saturierende Pushes stattfinden. Wir benutzen wieder die Potentialfunktion $\Phi = \sum_{v \in V \setminus \{s\}} e(v) \cdot d(v)$. Am Anfang von jeder Phase ist $\Phi \leq 4n^2\Delta$, nach den Lemmata 3 und 6. Eine Relabel-Operation erhöht Φ um höchstens 2Δ, wieder nach Lemma 6, so daß die höchstens $2n^2$ Relabel-Operationen innerhalb einer Phase Φ um höchstens $4n^2\Delta$ erhöhen. Jeder nicht-saturierende Push hat die Größe genau Δ und verringert also Φ um mindestens Δ. Daher gibt es innerhalb jeder Phase höchstens $(4n^2\Delta + 4n^2\Delta)/\Delta = 8n^2$ nicht-saturierende Pushes. ∎

Lemmata 4, 5 und 7 ergeben zusammen

Satz 1: Der skalierende Algorithmus bestimmt einen maximalen Fluß in einem Netzwerk mit n Knoten, m Kanten und maximaler Kantenkapazität U unter Ausführung von höchstens $nm + 10n^2 + 8n^2 \log_2 U$ Push- und Relabel-Operationen.

Die genaue Implementierung des skalierenden Algorithmus soll hier nicht erörtert werden. Insbesondere wird nicht beschrieben, wie aktive Knoten und wählbare Kanten effizient gefunden werden können. Man kann aber diese Probleme recht einfach lösen und erhält dadurch ein Verfahren, dessen Laufzeit tatsächlich proportional zu der in Satz 1 angegebenen Operationszahl ist.

5. Der inkrementelle skalierende Algorithmus

Bis auf geringfügige Verbesserungen war Satz 1 bis 1989 das beste bekannte Ergebnis für das hier betrachtete Problem. Insbesondere hielten viele nm für eine “magische Grenze”: Es wurde vermutet, daß kein Algorithmus mit weniger als nm Pushes auskommen kann. Daß diese Vermutung falsch ist, zumindest wenn m etwas größer als n und U nicht außerordentlich groß ist, wurde in den Arbeiten (Cheriyan and Hagerup, 1989) und (Cheriyan, Hagerup and Mehlhorn, 1990) gezeigt. Wir skizzieren hier eine der wesentlichen neuen Ideen in diesen Arbeiten.

Will man die Anzahl der Pushes unter nm drücken, muß man offensichtlich etwas gegen die saturierenden Pushes unternehmen. Dabei stören nur ganz kleine Pushes, denn größere Pushes verringern das Potential Φ wesentlich und können daher wie die nicht-saturierenden Pushes gezählt werden. Aber ganz kleine Pushes tragen nur unwesentlich zum Fortschritt des Algorithmus bei, so daß man versucht sein könnte, sie zu eliminieren.

Es reicht natürlich nicht, vor jedem Push einen Test auszuführen um festzustellen, ob der Push zu klein wäre, in welchem Fall er nicht ausgeführt wird, denn der Test wäre im wesentlichen so zeitaufwendig wie der Push selbst. Besser wäre es, wenn die Kante, die einen kleinen Push aufnehmen sollte, erst gar nicht vorhanden wäre. Kleine saturierende Pushes verlaufen im allgemeinen

über Kanten mit geringer Kapazität. Der Algorithmus aus (Cheriyan, Hagerup and Mehlhorn, 1990) geht daher inkrementell vor: Gestartet wird der Algorithmus auf einem Teilnetzwerk, das nur die Kanten mit den größten Kapazitäten enthält. Die restlichen Kanten werden nach und nach in der Reihenfolge abnehmender Kapazität in das momentane Netz aufgenommen, wenn ihre Kapazitäten, gemessen an dem aktuellen Δ, gerade so groß geworden sind, daß die Kanten "Beachtung verdienen". Dabei ist Vorsicht geboten, denn neu einzufügende Kanten können durchaus steil abfallen, so daß die Legalität der aktuellen Beschriftung gefährdet ist. Solche Kanten müssen sofort saturiert werden. Der Algorithmus geht so vor, daß jeder Knoten eine "versteckte" Reserve an Überschuß aufbaut, mit der neu eingefügte Kanten saturiert werden können, die aber nicht für normale Push-Operationen zur Verfügung steht. Knoten, die noch nicht genügend Reserveüberschuß sammeln konnten, dürfen die tiefste Ebene nicht verlassen (ihr d-Wert muß 0 bleiben), so daß sie nie in die Verlegenheit kommen können, steil abfallende Kanten saturieren zu müssen.

Wir verzichten auf die recht komplizierte Analyse von diesem Algorithmus und begnügen uns damit, das Ergebnis zu zitieren.

Satz 2: Es gibt eine Konstante $C > 0$, so daß der inkrementelle skalierende Algorithmus einen maximalen Fluß in einem Netzwerk mit n Knoten, m Kanten und maximaler Kantenkapazität U unter Ausführung von höchstens $C(n^{3/2}m^{1/2} + n^2 \log_2 U)$ Push- und Relabel-Operationen berechnet.

Für $m \approx n$ liefern die Sätze 1 und 2 bis auf einen konstanten Faktor die gleiche Operationszahl. Für den rechenintensiveren anderen Extremfall $m \approx n^2$ liefert aber Satz 2 eine Schranke proportional zu $n^{2.5}$, wenn U nicht außerordentlich groß ist, während die entsprechende Zahl für Satz 1 proportional zu n^3 ist. Der Unterschied ist ein sehr ins Gewicht fallender Faktor von $\sqrt{n}$. Die eigentliche Laufzeit des inkrementellen skalierenden Algorithmus, obwohl besser als für den einfachen skalierenden Algorithmus, ist allerdings leider nicht proportional zur Anzahl der Push- und Relabel-Operationen: Diese Anzahl ist so stark verringert worden, daß jetzt das Auffinden von wählbaren Kanten der Flaschenhals ist.

Literatur

AHUJA, R. K. AND ORLIN, J. B. (1987), A Fast and Simple Algorithm for the Maximum Flow Problem, Sloan W.P. No. 1905–87 (revised), MIT, October 1988.

CHERIYAN, J. AND HAGERUP, T. (1989), A Randomized Maximum-Flow Algorithm, Proceedings, 30th Annual Symposium on Foundations of Computer Science, pp. 118–123.

CHERIYAN, J., HAGERUP, T. AND MEHLHORN, K. (1990), Can a Maximum Flow be Computed in $o(nm)$ Time?, Proceedings, 17th International Colloquium on Automata, Languages, and Programming.

GOLDBERG, A. V. AND TARJAN, R. E. (1988), A New Approach to the Maximum-Flow Problem, *J. ACM* **35**, pp. 921–940.

TARJAN, R. E. (1983), Data Structures and Network Algorithms, SIAM, Philadelphia, Penn.