Stephan Rau

Ein experimenteller Vergleich von zwei Algorithmen zur Berechnung des maximalen Flusses in einem asymmetrischen Netzwerk mit reellen Kapazitäten

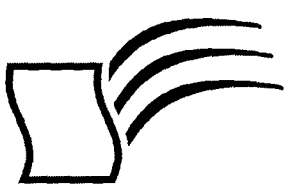
Diplomarbeit

Diplom.de

Stephan Rau

Ein experimenteller Vergleich von zwei Algorithmen zur Berechnung des maximalen Flusses in einem asymmetrischen Netzwerk mit reellen Kapazitäten

Diplomarbeit
an der Universität des Saarlandes
Januar 1997 Abgabe



***Diplomarbeiten* Agentur**
Dipl. Kfm. Dipl. Hdl. Björn Bedey
Dipl. Wi.-Ing. Martin Haschke
und Guido Meyer GbR

Hermannstal 119 k
22119 Hamburg

agentur@diplom.de
www.diplom.de

ID 373

ID 373
Rau, Stephan: Ein experimenteller Vergleich von zwei Algorithmen zur Berechnung des maximalen Flusses in einem asymmetrischen Netzwerk mit reellen Kapazitäten / Stephan Rau - Hamburg: Diplomarbeiten Agentur, 1997
Zugl.: Saarbrücken, Universität, Diplom, 1997

Dipl. Kfm. Dipl. Hdl. Björn Bedey, Dipl. Wi.-Ing. Martin Haschke & Guido Meyer GbR
Diplomarbeiten Agentur, http://www.diplom.de, Hamburg
Printed in Germany

DIPLOMARBEIT

Thema

Ein experimenteller Vergleich von zwei Algorithmen zur Berechnung des maximalen Flusses in einem asymmetrischen Netzwerk mit reellen Kapazitäten

Angefertigt von

Stephan Rau

am Fachbereich Informatik
der Universität des Saarlandes

1997

Vorbemerkungen

Die Disziplin "Effiziente Algorithmen" ist ein gutes Beispiel dafür, wie sich ein Teilbereich einer Wissenschaft innerhalb von fünfzehn bis zwanzig Jahren aus einer eher als Randlage zu bezeichnenden Situation zu einem zentralen Themenbereich in fast allen Bereichen der Wirtschaft und Technik entwickelt hat.

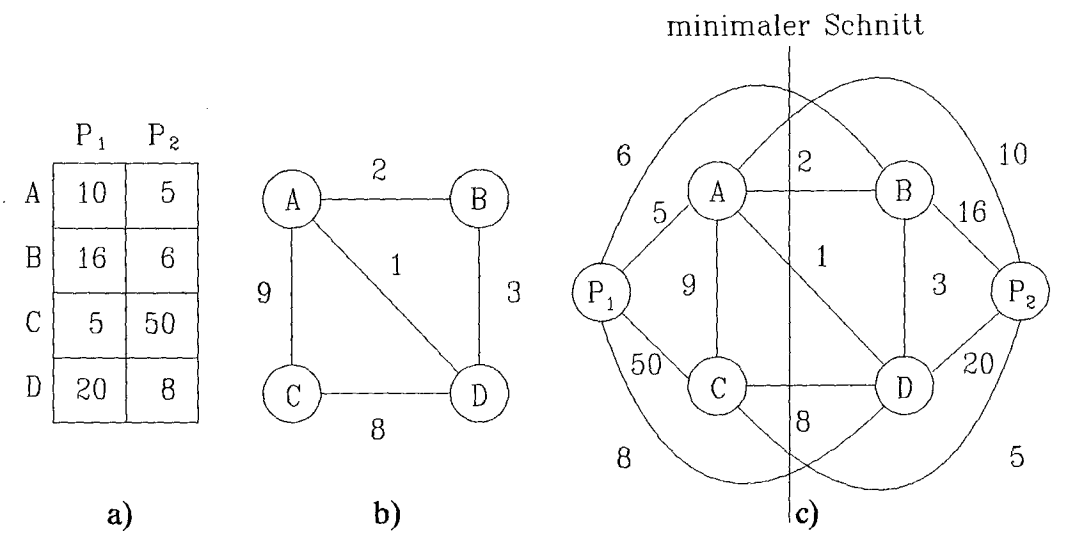
Als ich anfangs der achtziger Jahre mein Betriebspraktikum in einer Software-Abteilung der SIEMENS AG in München absolvierte, war die Situation nach meinen Eindrücken noch die, daß man auf Großrechnern mit im Verhältnis zur Komplexität des zu lösenden Problems gewaltig erscheinenden Ressourcen an Speicherplatz und Rechenzeit Programme entwickelte, die vor allem "funktionieren" mußten. Effizienzbetrachtungen wurden lediglich dann angestellt, wenn es galt, extrem komplexe Probleme aus der Informatik selbst wie beispielsweise die Kompaktifizierung der Interndarstellung von VLSI-Layouts für realistische Datenmengen auf eine Machbarkeitsstufe zu zwängen.

Der absolute Zwang hin zur theoretischen Modellierung nach Effizienzgesichtspunkten und weg von "ad hoc"-Lösungen hat auf breiterer Ebene unter anderem aus folgenden Gründen im oben beschriebenen Zeitintervall Raum gegriffen: man hat erkannt, daß selbst minimalste strukturelle Änderungen an Verfahrensweisen zu hohen Kostensenkungen führen können - so hat, als Beispiel am Rande erwähnt, vor wenigen Jahren bei den FORD-Werken in Saarlouis die Neuverlegung der Benzinleitung im Motorraum des Ford-Escort Modells dazu geführt, daß neben dem leichteren Einbau einige Plastikclips eingespart wurden, was alles in allem eine Ersparnis von ca. dreihunderttausend Mark im Jahr brachte (was natürlich immer noch sehr wenig für einen Konzern dieser Größenordnung ist); ein anderer Grund besteht darin, sowohl dem Bedürfnis als auch der Notwendigkeit nach immer komplexerem Informationsfluß selbst in Bereichen, die man früher nur schwerlich mit der Informatik in Verbindung bringen konnte, nachzukommen.

Das Beispiel schlechthin, das selbst der Normalverbraucher jeden Tag vor Augen hat, ist die Entwicklung auf dem Automobilsektor. War es ab Mitte der achtziger Jahre bei Fahrzeugen der gehobenen Klasse so, daß man im Zuge der Miniaturisierung auf Hardwareebene die ersten Motorsteuerungen mittels eines zentralen Computers in Serie auf den Weg brachte, so genügt das heute bei weitem nicht mehr. Heute ist es durchaus Standard, daß in einem Automobil bis zu fünf oder sechs Steuergeräte (für Motorsteuerung, Automatikgetriebesteuerung, Dieselregelung, Antiblockiersystem, Antischlupfregelung usw. . . um nur einige zu nennen) miteinander kommunizieren müssen und fünf bis zehn Kilometer an Verdrahtung verlegt werden müssen, um die einzelnen Komponenten miteinander zu verbinden. Dies hat zur Folge, daß man sich selbst in solch einem äußerlich relativ überschaubaren Bereich darüber Gedanken machen muß, wie man Informationsflüsse effizient gestaltet, d.h., man muß unter anderem wissen, welche Informationsmenge man zu einem gewissen Zeitpunkt über eine gewisse Komponente schicken kann (vgl. [11]).

So ist es nicht weiter verwunderlich, daß im Zuge der formalen Beschreibung solcher Problemstellungen für die Praxis die Berechnung des **maximalen Flusses** in einem Graphen ein klassisches Problem sowohl der Informatik als auch des Operations Research darstellt. Viele andere Problemstellungen aus der Technik können auf dieses Problem zuückgeführt werden. Im Bereich des Operations Research etwa löst man Transportprobleme oder die Planung und Steuerung von Projektabläufen für die industrielle Fertigung, indem man sie als Flußprobleme interpretiert (vgl. [7]). Sämtliche Problemstellungen, bei denen man sich in irgendeiner Form die Frage stellen muß, wieviele Einheiten eines bestimmten Gutes in einem bestimmten Zeitintervall über gewisse Transportwege - man denke dabei an Eisenbahnnetze, Rohrleitungssysteme, die Planung von Speditionsrouten, die Planung von Fließbandstraßen für Autos usw. . . . - geschleust werden können, führen letztendlich auf die Lösung eines Flußproblems zurück.

Bild 1 vgl. [6]>10.2.2.>S 463



a) Laufzeiten der Moduln auf den beiden Prozessoren
b) Graph für die Kommunikationszeiten der Moduln
c) Um die Prozessoren erweiterter Graph mit minimalem Schnitt

Ein anderes Beispiel (vgl. [6]>2.2.4.>S 47-54) liegt in der Planung von Computer- oder sonstigen elektronischen Netzwerken wie etwa Telefonnetzen. Hier ist es unter anderem wichtig, zuverlässige Verbindungen mit hoher Ausfalltoleranz herzustellen, d.h., eine Verbindung zwischen zwei Knoten X und Y (in Netzwerken "interface message processors" genannt) herzustellen, die auch dann noch funktioniert, wenn mehrere Verbindungswege abgerissen sind. Dieses Problem kann auf die Berechnung der Anzahl der kantendisjunkten Pfade in einem Netzwerk zurückgeführt werden, welche mit einem Algorithmus zur Berechnung des maximalen Flusses erfolgen kann.

Das letzte Beispiel (vgl. [6]>10.2.2.>S 462-464) wollen wir etwas ausführlicher darstellen (dazu gehört Bild 1). Beim Aufbau von Computernetzwerken, die so strukturiert sind, daß ein Zentralrechner mit hoher Rechenleistung mit mehreren Kleincomputern verbunden ist, entsteht das Problem, wie man ein Programm, das insgesamt zu groß ist, um auf einem Rechner (einschließlich Zentralrechner) laufen zu können, so auf die Rechner verteilt, daß es ablauffähig ist. Es wird nun an einem relativ konkreten Fallbeispiel gezeigt, wie man dieses Problem in ein Flußproblem überführt. Wir nehmen an, ein Programm bestünde aus vier Moduln A,B,C und D. Für jeden Modul sei die Laufzeit auf den Prozessoren P_1 und P_2 bekannt. Zusätzlich sei die Zeit für den Informationsaustausch (d.h. Zeit für Parameterübergabe usw.) zwischen einigen der Moduln bekannt. Dieser Sachverhalt wird durch Teil b) von Bild 1 wiedergegeben. Der Graph von Teil b) wird nun so erweitert, daß von P_1 (P_2) zu jedem Knoten eine Kante gezogen wird, welche mit der Laufzeit des betreffenden Moduls auf P_2 (P_1) beschriftet wird. Dann erhält man durch die Berechnung eines minimalen (P_1 , P_2) Schnittes (was gleichbedeutend zur Berechnung des maximalen Flusses ist, wie wir später sehen werden) eine optimale Zuordnung der Moduln zu den Prozessoren.

Die Netzwerke, für die man maximale Flüße berechnen will, können in unterschiedlichen Varianten vorliegen: die Kapazitäten der Kanten können ganzzahlig sein; die Kapazitäten können auf ein bestimmtes Werteintervall begrenzt sein; das Netzwerk kann planar sein; die Kanten können zudem mit einer Kostenfunktion bewertet sein usw. . .

In der vorliegenden Arbeit wird das Problem der Berechnung eines maximalen Flusses in einem gerichteten Netzwerk mit nichtnegativen, reellwertigen Kantenkapazitäten betrachtet und eine PASCAL-Implementierung für ein asymmetrisches Netzwerk mit diesen Eigenschaften angegeben. Das Hauptaugenmerk liegt dabei in der Betrachtung verschiedener Methoden zur Berechnung einer wesentlichen Teilaufgabe des Gesamtproblems, nämlich der Berechnung **blockierender Flüsse**. Es werden zwei grundsätzlich verschiedene Verfahren hierzu angegeben, aus welchen dann jeweils zwei Implementierungen fließen. Insgesamt werden folglich die Laufzeiten von vier Implementierungen verglichen.

Zum Abschluß dieses Vorwortes möchte ich mehreren Personen meinen Dank aussprechen: zunächst bedanke ich mich bei Herrn Professor Mehlhorn für die unkonventionelle und zugleich ermutigende Akzeptanz, die er mir entgegengebracht hat; desweiteren danke ich meinen Eltern dafür, daß sie mir ein Studium an einer Hochschule ermöglicht haben und meiner Schwester Christiane fürs Korrekturlesen; mein besonderer Dank gilt Frau Gertrud Kolling für ihre immerwährende Unterstützung.

Literaturliste

[1] Mehlhorn, K :
"Effiziente Algorithmen", Teubner-Verlag, 1977

[2] Mehlhorn, K :
"Datenstrukturen und effiziente Algorithmen, Band 1: Sortieren und Suchen", Teubner-Verlag, 1986

[3] Mehlhorn, K :
"Data Structures and Algorithms 2: Graph Algorihms and NP-Completeness", Springer-Verlag, 1984

[4] Mehlhorn, K :
"Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry", Springer-Verlag, 1984

[5] Ford, Fulkerson :
"Flows in Networks". Princeton University Press, 1962

[6] Tanenbaum :
"Computer Networks", Prentice-Hall, 1981

[7] Gal :
"Grundlagen des Operations Research 2", Springer-Verlag, 1987

[8] Malhotra, Kumar, Maheshwari :
"An O($|V|^3$) Algorithm for Finding Maximum Flows in Networks", IPL, vol 7, 277-278, 1978

[9] Dinic :
"Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation", Soviet Math. Dokl. 11, 1277-1280, 1970

[10] Galil, Naamad :
"Network Flow and Generalized Path Compression", 11th STOC, 13-26, 1979

[11] Robert Bosch Gmbh :
"Controller Area Network", Technische Unterrichtung Kraftfahrzeugausrüstung-Vorentwicklung Systeme, 1987

Inhaltsverzeichnis

1. GRUNDLAGEN AUS DER GRAPHENTHEORIE

In diesem einleitenden Kapitel werden einige grundlegende, wohlbekannte Tatsachen aus der Graphentheorie zusammengestellt (vgl. z.b. [3]>IV.1.). Zunächst eine Definition, die behilflich ist, um asymptotische Aussagen über Laufzeit und Speicherplatz treffen zu können.

Definition 1 vgl. [2]>I.6.>S 33

Sei $f : \mathbb{N} \to \mathbb{R}$ eine Funktion. Dann bezeichnen $O(f)$ sowie $\Omega(f)$ folgendes:

a) $O(f) = \{ g : \mathbb{N} \to \mathbb{R} \mid \exists\, c > 0\ \exists\, n_0$ mit $g(n) \leq c \cdot f(n)$ für alle $n \geq n_0 \}$

oder, falls $f(n) \neq 0$ für fast alle $n \in \mathbb{N}$:

$O(f) = \{ g : \mathbb{N} \to \mathbb{R} \mid \exists\, c > 0$ mit $g(n) \leq c + c \cdot f(n)$ für alle $n \in \mathbb{N} \}$

b) $\Omega(f) = \{ g : \mathbb{N} \to \mathbb{R} \mid \exists\, c > 0\ \exists\, n_0$ mit $g(n) \geq c \cdot f(n)$ für alle $n \geq n_0 \}$

▨

Bemerkung 1

---

a) Wir nehmen als zugrundeliegendes Rechnermodell eine RAM unter dem Einheitskostenmaß an (vgl. [2]>I.1.).

b) "log" bezeichnet immer den Logarithmus zur Basis 2.

▨

Definition 2 vgl. [3]>IV.1.> S 1

Ein gerichteter Graph $G = (V , E)$ besteht aus einer Knotenmenge $V = \{ 1 , \ldots , |V| \}$ und einer Kantenmenge $E \subseteq V \times V$. Ein Paar $(u , v) \in E$ heißt eine Kante von u nach v. Weiterhin sind $n = |V|$ und $e = |E|$.

▨

Da die Matrixdarstellung eines Graphen immer $\Omega(n^2)$ Speicherplatz benötigt, implementieren wir Graphen durch eine Adjazenzlistendarstellung, welche lediglich Speicherplatz $O(e + n)$ benötigt.

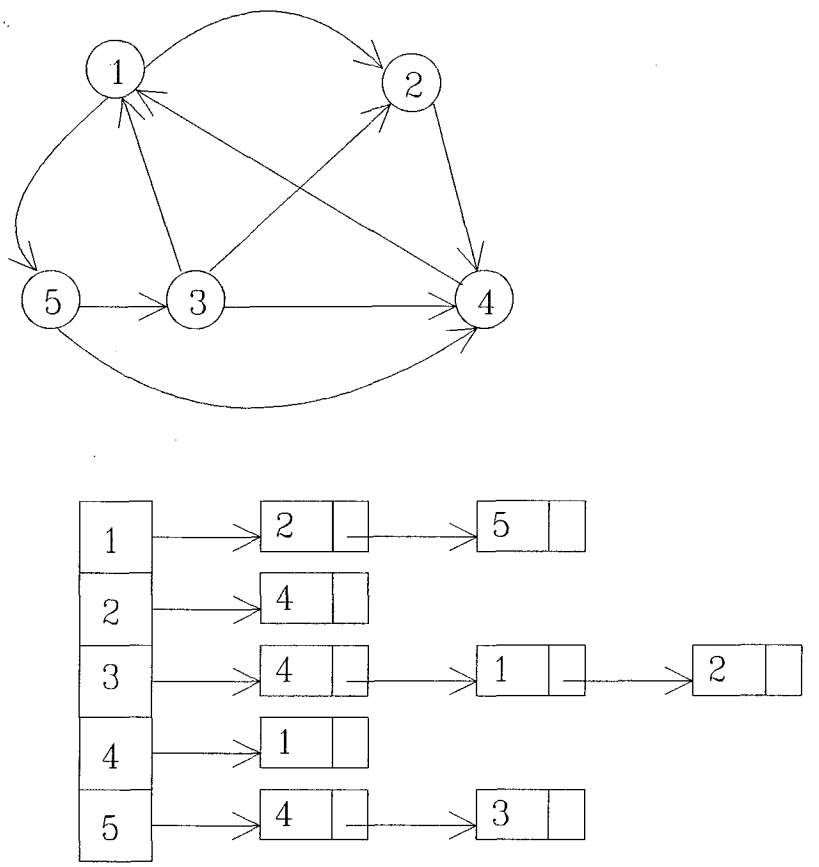
Definition 3 vgl. [3]>IV.1.>S 2-3

Eine Adjazenzlistendarstellung eines Graphen besteht aus n linearen Listen. Die i-te Liste enthält alle Knoten j mit $(i , j) \in E$.

▨

Wir erhalten somit in PASCAL-Notation die in Prog 1 angegebene Implementierung eines Graphen. Die Punktierung steht für Komponenten, die je nach Problemstellung noch hinzuzufügen sind. Bild 2 zeigt einen Graphen mit zugehöriger Adjazenzliste.

Bild 2



Graph mit zugehöriger Adjazenzlistendarstellung

Prog 1

```
type node = record name : 1 . . n;
                      .
                      .
                   next : ^node;
            end;
type adjhead = array [1 . . n] of ^node;
```

Definition 4 vgl. [3]>IV.1.>S 3

Sei $G = (V , E)$ ein gerichteter Graph. Dann gilt:

a) Ein Pfad von v nach w, wobei $v , w \in V$, ist eine Folge $v_0 , v_1 , . . , v_k$ von Knoten, sodaß $v = v_0 , w = v_k$ und $(v_i , v_{i+1}) \in E$ für $0 \leq i < k$. Dabei ist k die Länge des Pfades.

b) Ein Pfad heißt einfach, falls $v_i \neq v_j$ für $0 \leq i < j < k$. Ein Kreis ist ein Pfad von v nach v.

c) Ein Pfad heißt azyklisch, falls er keine Kreise enthält.

d) Ein Graph heißt ungerichtet, falls gilt: $(u , v) \in E \Leftrightarrow (v , u) \in E$.

e) Ein Graph heißt zusammenhängend, falls der unterliegende ungerichtete Graph zusammenhängend ist.

f) Ein ungerichteter Graph heißt zusammenhängend, falls für jedes Paar $v , w \in V$ ein Pfad von v nach w existiert.

g) $indeg_G(v) := | \{ w | (w , v) \in E \} |$;
$outdeg_G(v) := | \{ w | (v , w) \in E \} |$.

h) Ein zusammenhängender, gerichteter, azyklischer Graph T, bei dem es einen einzigen Knoten $r \in V$ mit $indeg_T(r) = 0$ gibt und für alle $w \in V, w \neq r$, $indeg_T(w) = 1$ gilt, heißt **Baum**.

Ein Knoten $b \in V$ heißt Blatt des Baumes, wenn $outdeg_T(b) = 0$ gilt. Den Knoten v mit $indeg_T(v) = 0$ nennt man die Wurzel des Baumes.

Definition 5

Sei $T = (V , E)$ ein Baum. Dann gilt:

a) Sei $v \in V$ und $W = \{ v \} \cup \{ w | \exists$ Pfad von v nach w in T $\}$, dann heißt der Baum $U = (W , E \cap (W \times W))$ Unterbaum von T.

b) Wenn $(u , v) \in E$ und $(u , w) \in E, v \neq w$, dann heißt v Schwester von w.

c) Wenn $(u , v) \in E$, dann heißt v Kind von u und u heißt Elter von v.

d) Tiefe(v , T) := Länge des Pfades von r nach v, wobei r Wurzel von T ist.

e) Höhe(T) := max{ Tiefe(v , T) | v ∈ V }.

▨

Die Lösungen der meisten Problemstellungen, denen eine graphische Darstellung zugrundeliegt, benötigen Algorithmen zur systematischen Durchmusterung eines Graphen. Im folgenden wird ein Algorithmus angegeben, der dies leistet. Dabei ist S die Menge der Knoten, die schon betrachtet wurden. SQ ist eine Teilmenge von S, die aus allen Knoten von S besteht, aus denen noch unbenutzte Kanten hinausgehen.

Prog 2

```
proc explorefrom( s );

begin S := { s };
      SQ := { s };
      while SQ ≠ ∅ do
      begin wähle ein v ∈ SQ;
            sei ( v , w ) die nächste unbenutzte Kante aus v heraus;
            if ( v , w ) nicht existiert then

            lösche v aus SQ
                                        else
            if v ∉ S then
            begin füge w zu S hinzu;
                  füge w zu SQ hinzu;
            end;
      end;
end;
```

Bemerkung 2

Falls bei der Prozedur explorefrom die Menge SQ als Stack (vgl. [2]>I.4.1.) implementiert wird, spricht man von einer **D**(epth)**F**(irst)**S**(earch) - Durchmusterung des Graphen.
Falls die Menge SQ als Queue (vgl. [2]>I.4.1.) realisiert wird, so spricht man von einer **B**(reath)**F**(irst)**S**(earch) - Durchmusterung des Graphen.
BFS-Durchmusterungen werden wir später in leicht modifizierter Form benötigen (vgl. Prozedur "bfs" in 2.2.), um bestimmte Teilgraphen aus dem Gesamtnetzwerk herauszufiltern.

▨

2. GRUNDLAGEN AUS DER FLUSSTHEORIE

Dieses Kapitel beinhaltet ein grundlegendes Verfahren zum Berechnen maximaler Flüsse in Netzwerken. Es besteht wie auch die folgenden Abschnitte aus einem theoretischen und einem praktischen Teil. Der praktische Teil wird immer aus einer PASCAL-Implementierung dessen, was im jeweils vorausgehenden Abschnitt des betreffenden Kapitels hergeleitet worden ist, bestehen. Nimmt man alle praktischen Abschnitte zusammen, so entsteht ein komplettes PASCAL-Programm.

2.1. Grundlegende Definitionen sowie ein Verfahren zur Berechnung maximaler Flüsse in Netzwerken

Dieser Abschnitt beginnt mit grundlegenden Definitionen aus der Netzwerktheorie. Das daran anschließend angegebene Verfahren zur Berechnung von maximalen Flüssen auf Netzwerken wird dann am Ende des Abschnitts anhand eines ausführlichen Beispiels erläutert.

Definition 6 vgl. [3]>IV.9.1.>S 59-60

Ein gerichtetes Netzwerk $N = (V , E , cap)$ besteht aus einem gerichteten Graphen $G = (V , E)$ und einer Funktion $cap : E \rightarrow \mathbb{R}_+ \cup \{ 0 \}$.
Seien $s , t \in V$ zwei ausgezeichnete Knoten, namentlich die Quelle s und das Ziel t. Eine Funktion $f : E \rightarrow \mathbb{R}$ heißt ein **zulässiger Fluß**, wenn sie folgende Bedingungen erfüllt:

a) $0 \leq f(e) \leq cap(e)$ für alle $e \in E$;

b) $\sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$ für alle Knoten $v \in V - \{ s , t \}$,

wobei $in(v)$ ($out(v)$) die Menge der in v endenden (von v ausgehenden) Kanten bedeutet.

Falls $f : E \rightarrow \mathbb{R}$ ein zulässiger Fluß ist, so ist

$val(f) = \sum_{e \in out(s)} f(e) - \sum_{e \in in(s)} f(e)$ der Flußwert von f.

Falls zusätzlich gilt $(u , v) \in E \Rightarrow (v , u) \notin E$, so nennt man N asymmetrisch.
▨

Das Problem des **maximalen Flusses** in einem gerichteten Netzwerk besteht nun darin, einen zulässigen Fluß mit maximalem Flußwert zu berechnen.

Definition 7 vgl. [3]>IV.9.1.>S 60

Ein (s , t) - Schnitt ist eine Aufteilung S , T von V mit $V \cap T = \emptyset$ und $S \cup T = V$. Die Kapazität des Schnittes ist gegeben durch

$$cap(S , T) = \sum_{e \in E \cap (S \times T)} cap(e) .$$

▨

Satz 1 vgl. [3]>IV.9.1.>S 69-70

Sei (N , E , cap) ein Netzwerk mit $s , t \in V$. Sei f_{max} der maximale Flußwert aller zulässigen (s , t) - Flüsse und cap_{min} die minimale Kapazität aller (s , t) - Schnitte. Dann gilt:

$$f_{max} = cap_{min}$$

▨

Dieser Satz gilt als ein Kernsatz der Flußtheorie. Leider ist die Aussage des Satzes nicht konstruktiv, d.h., der Satz weist uns keinen (effizienten) Weg, wie ein maximaler Fluß tatsächlich gefunden werden kann.

Wir kommen nun zu einem wesentlichen Verfahren zur Bestimmung maximaler Flüsse in Netzwerken. Dieses Verfahren und alles in diesem Abschnitt folgende geht im wesentlichen auf Ford/Fulkerson (vgl. [5]) zurück. Das Verfahren basiert darauf, daß man, mit dem Nullfluß startend, ausgehend vom Knoten s flußvergrößernde Pfade sucht. Dabei betrachtet man in einem Durchlauf nur die kürzesten Pfade (bezogen auf die Kantenanzahl) von s nach t. Da es manchmal dabei auch notwendig ist, den schon bestehenden Fluß an einer Kante im Interesse einer Vergrößerung des Gesamtflusses im Netz zu verringern, ist die folgende Definition sinnvoll.

Definition 8 vgl. [3]>IV.9.1.> S 61-62

a) $E_1 := \{ (v , w) \mid (v , w) \in E \text{ und } f(e) < cap(e) \}$;

b) $E_2 := \{ (w , v) \mid (v , w) \in E \text{ und } f(e) > 0 \}$;

c) $lncap : E_1 \cup E_2 \to \mathbb{R}$ mit $\begin{cases} lncap(e_1) = cap(e) - f(e) & \text{für } e_1 \in E_1 \\ lncap(e_2) = f(e) & \text{für } e_2 \in E_2 \end{cases}$

d) $V_0 := \{ s \}$;

$V_{i+1} := \{ w \in V - \{ V_1 \cup .. \cup V_i \} \mid \exists v \in V_i (v , w) \in E_1 \cup E_2 \}$ für $i \geq 0$;

$\overline{V} := \bigcup_{i \geq 0} V_i$;

e) L(ayered)N(etwork) := $(\overline{V} , (E_1 \cup E_2) \cap \bigcup_{i \geq 0} (V_i \times V_{i+1}) , lncap)$,

kurz LN = ($\overline{V}$, $\overline{E}$, lncap) wobei $\overline{E}$ die Kantenmenge in der darüberliegenden Zeile bedeutet.

▨

Wir können nun unseren Basis-Datentyp aus Prog 1 (Seite 2) gemäß Definition 8 auffüllen. Die "inln"-Komponente gibt an, ob die betreffende Kante in LN aufgenommen wurde und somit an der Berechnung eines blockierenden Flusses teilnimmt. Über die "realp"-Komponente geschieht das Updating mit Kanten aus E_2 (vgl. Prozedur "eval" in 2.2.). Die "intree"-Komponente gibt an, ob die bei der Berechnung eines blockierenden Flusses gerade betrachtete Kante schon Teil eines Pfades (Baumes) ist (Bedeutung wird erst im nächsten Kapitel klar). Die Bedeutung der restlichen Komponenten wird unmittelbar aus Definition 10 klar.

Prog 3

```
type node = record
                    name : integer;
                     cap : real;
                    flow : real;
                   lncap : real;
                  lnflow : real;
                    inln : boolean;
                   realp : ^node;
                  intree : ^edgelist;
              next , last : ^node;
            end;
```

Der in Prog 4 beschriebene Basis-Algorithmus berechnet aus dem Ausgangsnetzwerk N fortgesetzt LN's, berechnet auf diesen einen blockierenden Fluß und addiert diesen zu dem schon vorhandenen Fluß in N. Dies tut er solange, bis kein Weg von s nach t mehr gefunden werden kann, der noch eine Flußvergrößerung zuläßt. Die Neuberechnung des Flusses im Ausgangsnetzwerk N und das Abbruchkriterium für den Algorithmus werden im folgenden Satz präzisiert.

Satz 2 vgl. [3]>IV.9.1.>S 62-63

Sei f ein zulässiger (s , t) - Fluß im Netzwerk N und sei
LN = ($\overline{V}$, $\overline{E}$, lncap) . Dann gilt:

a) f ist ein maximaler Fluß $\Leftrightarrow$ $t \notin \overline{V}$

b) Sei f ein zulässiger (s , t) - Fluß in LN. Dann ist $f : E \to \mathbb{R}$ mit

$\tilde{f}(e) = f(e) + \overline{f}(e_1) - \overline{f}(e_2)$

ein zulässiger Fluß in N mit Flußwert $val(f) + val(\overline{f})$.

▨

Prog 4 vgl. [3]>IV.9.1.>S 64

```
for all e ∈ E do f( e ) := 0;

konstruiere LN aus f;

while t ∈ V̄ do
begin
        finde blockierenden Fluß in LN;
        berechne f aus f̄ gemäß Satz 2 neu;
        konstruiere LN aus f;
end;
```

Es ist klar, daß die Berechnung von LN in Zeit O(e + n) erfolgen kann. Für die Anzahl der Schleifendurchläufe gilt das folgende Lemma.

Lemma 1 vgl. [3]>IV.9.1.> S 64-65

Der in Prog 4 angegebene Algorithmus benötigt O(n) Schleifendurchläufe.

▨

Definition 9 vgl. [3]>IV.9.1.> S 64

Ein zulässiger Fluß $\overline{f}$ in LN heißt blockierend, falls für jeden Pfad

$s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots \xrightarrow{e_k} v_k = t$ von s nach t mindestens eine Kante abgesättigt ist,

d.h. $\overline{f}(e_i) = lncap(e_i)$ für mindestens ein i, $1 \le i \le k$.

▨

Es bleibt die Aufgabe, ein effizientes Verfahren zur Berechnung eines blockierenden Flusses zu finden, was im nächsten Kapitel erfolgt.

Bemerkung 3

Aus der Konstruktion von LN ergeben sich folgende Eigenschaften: