```
void MF_Labeling(const graph& G, node s, node t, const
   edge_array<int>& cap, edge_array<int>& flow){
     list<node> L; \\ Menge S
     node_array<bool> labeled(G, false);
     node_array<edge> PRED(G, NULL);
     while(true){
          labeled[s] = true;
          L.append(s);
          while(!L.empty()){
               node v = L.pop();
               edge e;
               forall_out_edges(e,v){
                    if(flow[e] == cap[e])
                         continue; // e nicht in G(x)
                    node w = G.target(e);
                    if(labeled[w])
                         continue;
                    labeled[w] = true;
                    PRED[w] = e;
                    L.append(w);
               }
               forall_in_edges(e,v){
                    if(flow[e] == 0) continue;
                    node w = G.source(e);
                    if(labeled[w]) continue;
                    labeled[w] = true;
                    PRED[w] = e;
                    L.append(w);
               }
               if(labeled[t])
                    L.clear();
          } // Ende while-Schleife
          if(labeled[t])
               AUGMENT(G,s,t,PRED,cap,flow);
          else
               break; // ex. kein erhöhender Pfad
     }
}
```

Listing 5.2: MF_Labeling

```
1  void AUGMENT(const graph& G, node s, node t, const
       node_array<edge>& PRED, const edge_array<int>& cap,
       edge_array<int>& flow){
2      int delta = MAXINT; // Restkapazität von P
3      node v = t;
4      while(v != s){
5            int r;
6            edge e = PRED[v];
7            if(v == G.source(e)){ //Rückwärtskante
8                  r = flow[e];
9                  v = G.target(e);
10           }
11           else{ // Vorwärtskante
12                 r = cap[e] - flow[e];
13                 v = G.source(e);
14           }
15           if(r < delta)
16                 delta = r; // min
17     }
18     // Eigentliche Flusserhöhung
19     v = t;
20     while(v != s){
21           edge e = PRED[v];
22           if(v == G.source(e)){ //Rückwärtskante
23                 flow[e] = flow[e] - delta;
24                 v = G.target(e);
25           }
26           else{ // Vorwärtskante
27                 flow[e] = flow[e] + delta;
28                 v = G.source(e);
29           }
30     }
31 }
```

Listing 5.3: Augment

**Erklärung:**
Die äußere *while*-Schleife läuft so lange, bis explizit aus ihr herausgesprungen wird. Dies geschieht, wenn nach der inneren Schleife $t$ nicht gelabelt ist, es also keinen erhöhenden Pfad mehr gibt.
Da $s$ der Startknoten ist und jeder von $s$ aus erreichte Knoten gelabelt und zur Men-

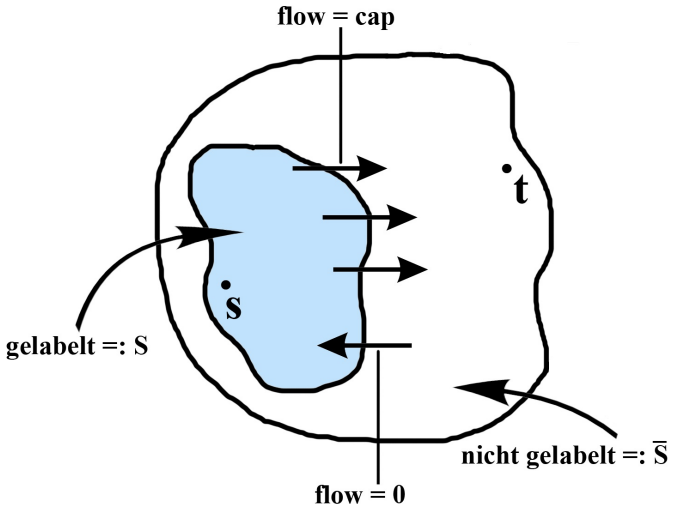
ge $L$ hinzugefügt wird, wird $s$ zu Beginn immer gelabelt und in L eingefügt. In der inneren Schleife wird zuerst ein beliebiger Knoten $u$ aus der Menge $L$ entnommen. Von diesem Knoten $u$ aus werden nun sowohl die ausgehenden, als auch die eingehenden Kanten betrachtet. Dabei werden diejenigen Kanten, die keine Restkapazität mehr besitzen, sofort aussortiert und nicht weiter betrachtet. Bei den restlichen wird überprüft, ob der durch die Kante erreichbare Knoten schon gelabelt wurde. Ist dies nicht der Fall wird er gelabelt, sein $PRED$-Verweis auf die Kante gesetzt, über die er erreicht wurde und der Knoten in die Menge $L$ aufgenommen.
Die innere Schleife bricht ab, sobald $L$ leer ist, was der Fall ist, wenn $t$ gelabelt wurde, da dann ein erhöhender Pfad existiert und entlang diesem, im Algorithmus AUGMENT, Fluss geschickt wird.
Hierbei wird zuerst über den gefundenen Pfad von $t$ nach $s$ gelaufen, und dabei die maximale Flusserhöhung ausfindig gemacht. Ist diese gefunden, also $s$ erreicht, wird erneut über den gefundenen Pfad von $t$ nach $s$ gelaufen und dabei die Flussänderung um den herausgefundenen Wert durchgeführt.
Es wird immer von $t$ nach $s$ gelaufen, da man nur den Vorgänger, nicht aber den Nachfolger eines Knotens kennt.

## Korrektheit

In jedem Schritt (Hauptschleife) gibt es zwei Möglichkeiten:

1. Der Algorithmus findet einen erhöhenden Pfad → AUGMENT
2. Der Algorithmus findet keinen erhöhenden Pfad. $t$ wird also nicht gelabelt → STOP

Zu zeigen: Im zweiten Fall ist der berechnete Fluss maximal.




$s \in S$, $t \in \bar{S}$
$\Rightarrow [S,\bar{S}]$ ist ein $(s,t)$-Schnitt.

**Beobachtung:**
$\forall\ (i,j) \in [S, \bar{S}]$ gilt: $u_i j = 0$ (deshalb werden diese Kanten vom Algorithmus nicht mehr benutzt, um weitere Knoten zu labeln).
Bei uns bedeutet das:
$\forall$ Kanten $e \in (S, \bar{S}) :\ flow[e] = cap[e]$ und
$\forall\ e \in (\bar{S}, S) :\ flow[e] = 0$

**Beobachtung über Flüsse und Schnitte**
Flusswert: $F = \sum\limits_{(i,j)\in(S,\bar{S})} x_{ij} - \sum\limits_{(i,j)\in(\bar{S},S)} x_{ij}$
hier gilt:
$$F = \underbrace{\sum_{(i,j)\in(S,\bar{S})} \overset{cap[e]}{u_{ij}}}_{u[S,\bar{S}]} - \sum_{(i,j)\in(\bar{S},S)} \overset{flow[e]}{0}$$
$\Rightarrow F = u[S, \bar{S}]$

Aus schwacher Version des MaxFlow/MinCut-Theorems ($F_{max} \leq \mathcal{U}_{min}$) folgt:
$\underbrace{F\text{ist maximal}}_{\Rightarrow \text{Korrektheit}}$ und $u[S, \bar{S}]$ ist minimal.

**Allgemein:**
Der Algorithmus liefert zusätzlich zur optimalen Lösung des primalen Problems (MaxFlow) eine optimale Lösung des sogenannten dualen Problems (MinCut). Dies bietet eine Möglichkeit das Ergebnis zu überprüfen (hier: MaxFlow = MinCut).

**Theorem 1** (MaxFlow/MinCut-Theorem)
*Der Wert eines maximalen Flusses ist gleich der Kapazität eines minimalen (s,t)-Schnittes.*

**Beweis**
Die Korrektheit des Theorems folgt aus der Korrektheit des Labeling-Algorithmus. □

**Theorem 2** (Augmenting-Path-Theorem)
*Ein Fluss $x$ ist genau dann maximal, wenn es im Restnetzwerk $G(x)$ keinen erhöhenden Pfad mehr gibt bzw. keinen Pfad mehr von $s$ nach $t$ gibt.*

**Beweis**

„ $\Rightarrow$ “ Falls $x$ maximal ist, existiert kein erhöhender Pfad mehr.

„ $\Leftarrow$ “ Falls kein erhöhender Pfad existiert, berechnet der Labeling-Algorithmus einen $(s,t)$-Schnitt $[S, \bar{S}]$ mit $F(x) = u[S, \bar{S}]$
$\Rightarrow x$ ist maximal.

□

**Theorem 3** (Ganzzahligkeit)
*Wenn alle Kapazitäten ganzzahlig ($u_{ij} \in \mathbb{N}$) sind, dann existiert ein maximaler Fluss $x$ mit $x_{ij} \in \mathbb{N} \; \forall \; (i,j) \in E$*

**Beweis**
Eine Flussänderung entlang von Kreisen in $G(x)$ ist möglich, ohne dabei den Flusswert $F_{max}$ zu verändern.
Sei $(i,j)$ eine Kante, sodass $x_{ij}$ nicht ganzzahlig ist. $i, j \notin \{s,t\}$
$\Rightarrow$ Mindestens eine Kante inzident zu $i$ und mindestens eine inzident zu $j$ in $G(x)$ hat ebenfalls ein nicht ganzzahliges $x$.
Das folgt aus der Massenbalance-Bedingung (5.1.1).
$\Rightarrow$ All diese Kanten liegen auf Kreisen.
Starte mit einer Kante, deren $x_{ij}$ nicht ganzzahlig ist.
Laufe dann solange über nicht ganzzahlige Nachbarkanten, bis der Kreis geschlossen ist. Der Kreis besitzt die Restkapazität $\delta$.
Eine Flusserhöhung bzw. -verminderung um $\delta$ macht mindestens ein $x_{ij}$ auf dem Kreis ganzzahlig. □

## Laufzeitanalyse

Sei $\mathcal{U} := max\{u_{ij} | (i,j) \in E\}$ obere Schranke für $F_{max}$.
Wir wissen, dass $F_{max} \leq u[S, \bar{S}] \; \forall \; (s,t)$-Schnitte $[S, \bar{S}]$ (MaxFlow/MinCut-Theorem)

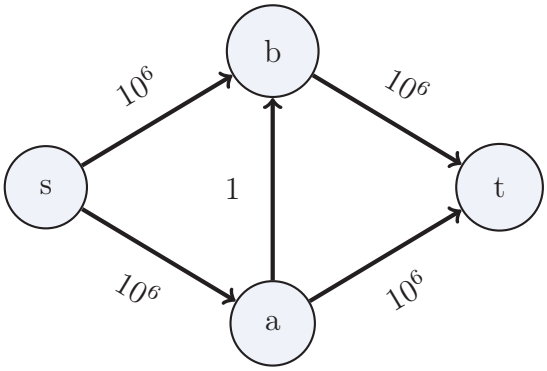
**Beobachtung:**
Höchstens $n-1$ Kanten verlaufen von $s$ nach $t$

$\Rightarrow u[S, \bar{S}] \leq (n-1) \cdot \mathcal{U}$
$\Rightarrow F_{max} \leq n \cdot \mathcal{U}$

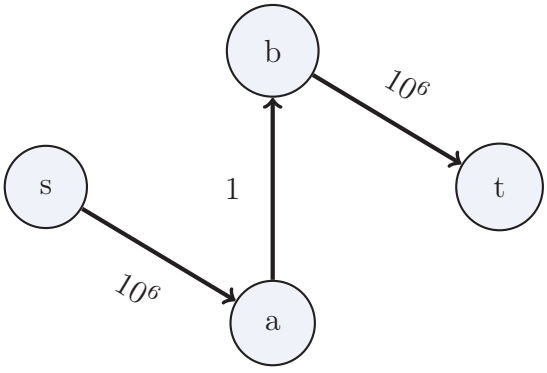
In jeder Iteration (AUGMENT) erhöht der Algorithmus den Flusswert um mindestens 1.
Daraus folgt, dass maximal $F_{max}$ Iterationen benötigt werden, also maximal $n \cdot \mathcal{U}$ viele Iterationen.
Eine Iteration (Labeling) kostet Zeit $\mathcal{O}(n+m)$ (siehe DFS, BFS ...)
$\Rightarrow$ Gesamtlaufzeit $\mathcal{O}(n^2 \cdot \mathcal{U} + m \cdot n \cdot \mathcal{U})$

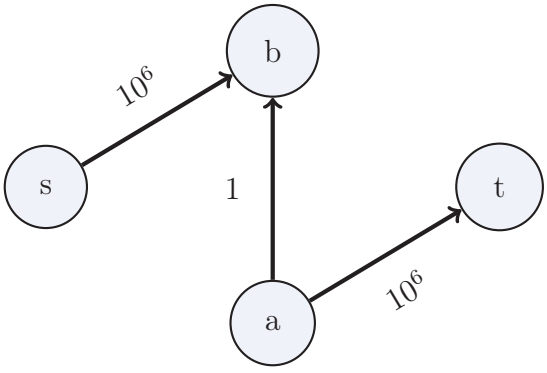
Annahme: $G$ ist zusammenhängend.
$\Rightarrow m \geq n-1$
$\Rightarrow$ Laufzeit $\mathcal{O}(n \cdot m \cdot \mathcal{U})$
nicht polynomielle Laufzeit in $m$ und $n$.

**Beispiel 5.2** (Worst-Case)




Abwechselnd existieren folgende erhöhende Pfade (mit jeweils $\delta = 1$)



**Verbesserungen:**

1. Versuche Pfade mit möglichst hoher Restkapazität zu finden. (→ Capacity Scaling)
2. Suche nach kürzesten erhöhenden Pfaden (# Kanten) (→ BFS)

**Laufzeiten von Algorithmen:**
Man kann Algorithmen in Bezug auf ihre Laufzeit verschiedenen Klassen zuordnen.

- Nicht polynomielle Algorithmen
  Die Laufzeit ist nicht polynomiell in $n$ und $m$.
  Z.B. der Labeling-Algorithmus ($n$ mal $\mathcal{U}$ Erhöhungen)
- Polynomielle Algorithmen
  Die Laufzeit ist polynomiell in $n$ und $m$ und $\log(\mathcal{U})$
  Z.B. Wortlänge des Rechners (z.B. 64 Bit)
- Streng polynomielle Algorithmen
  Die Laufzeit ist nur in n und m polynomiell

### 5.4.3 Algorithmus Capacity-Scaling

Capacity-Scaling ist ein polynomieller Algorithmus für MaxFlow.

**Idee:**
Versuche erhöhende Pfade mit großer Restkapazität ($\delta$) zu finden.
Beginne mit $\underbrace{\mathcal{U}, \text{ dann } \mathcal{U}/2,\ \mathcal{U}/4, \ldots 1}_{log(\mathcal{U}) \text{ Phasen } \rightarrow \Delta\text{-Phasen}}$

**Modifikation des Labeling-Algorithmus:**
Führe einen neuen Parameter $\Delta$ ein.
→ Ignoriere alle Kanten $(i, j)$ in $G(x)$ deren Restkapazität $r_{ij} < \Delta$

Dadurch ändert sich ausschließlich das Ausfiltern der Kanten (5.3).

```
forall_out_edges(e,v){
      if(cap[e] - flow[e] < delta)
            continue;
      //Rumpf
}
forall_in_edges(e,v){
      if(flow[e] < delta)
            continue;
      //Rumpf
}
```

Listing 5.4: Ausfiltern der Kanten

Der neue Parameter muss dem Algorithmus MF_Labeling noch zusätzlich übergeben werden.
→ MF_Labeling($G$, $s$, $t$, $cap$, $flow$, $\Delta$)

**Beobachtung:**
Für $\Delta = 1$ entspricht der Algorithmus dem normalen Labeling-Algorithmus (ganzzahlig!).

```
Capacity_Scaling(const graph& G, node s, node t, const
   edge_array<int>& cap, int U){
      edge_array<int>& flow(G,0);
      int delta = 2^log(U);
      while(delta < 0){
            MF_Labeling(G, s, t, cap, flow, delta)
            delta = delta/2
      }
}
```

Listing 5.5: Capacity_Scaling

**Andere Betrachtungsweise:**
$\Delta$-Restnetzwerk $G(x, \Delta)$ ist $G(x)$ ohne alle Kanten mit $r_{ij} < \Delta$.
Jede $\Delta$-Phase sucht einen erhöhenden Pfad in $G(x, \Delta)$.

## Korrektheit

Die letzte $\Delta$-Phase ($\Delta = 1$) entspricht dem normalen Labeling-Algorithmus.

## Laufzeit

Die Anzahl der Phasen (Hauptschleife) ist kleiner gleich $log(\mathcal{U})$.

**Lemma 5**
*Jede $\Delta$-Phase führt maximal $2 \cdot m$ Erhöhungen aus.*

**Beweis**
Betrachte das Ende einer $\Delta$-Phase.
Sei $S$ die Menge der gelabelten Knoten und $\bar{S} = V \setminus S$.
Dann ist $[S, \bar{S}]$ ein $(s,t)$-Schnitt ($s \in S$, $t \in \bar{S}$).
Die Restkapazität $r[S, \bar{S}] \leq m \cdot \Delta$, weil jede Kante zwischen $S$ und $\bar{S}$ höchstens (eigentlich $<$) Restkapazität $\Delta$ hat.
Betrachte die (nächste) $\Delta/2$-Phase.
Diese führt dann maximal $\frac{m \cdot \Delta}{\Delta/2}$ Erhöhungen aus.

$$\leq 2 \cdot m \text{ Erhöhungen.}$$

$\Rightarrow$ Laufzeit einer einzelnen $\Delta$-Phase ist $\mathcal{O}(\underbrace{2m}_{Anzahl} \cdot \underbrace{m}_{\substack{1x \\ Labeling}})$

□

**Lemma 6**
*Capacity_Scaling löst das MaxFlow-Problem in Zeit $\mathcal{O}(m^2 \cdot log(\mathcal{U}))$.*

**Beweis**
Eine $\Delta$-Phase kostet $\mathcal{O}(m^2)$ und die Anzahl der Phasen beträgt $log(\mathcal{U})$. □

**Bemerkung:**
Eine andere Möglichkeit den Labeling-Algorithmus zu verbessern ist, dass man kürzeste (# Kanten) Pfade verwendet ($\rightarrow$ Ford/Fulkerson)
und nur Kanten zwischen benachbarten Levels nutzt.

## 5.5 Preflow-Push

**Idee:**
Sende von $s$ aus möglichst viel Fluss ins Netzwerk $k$ und versuche ihn schrittweise bis zum Knoten $t$ zu leiten. Lasse dabei Überschuss wieder zurück nach $s$ fließen. Die Richtung findet man mit Hilfe von Distanz-Labels.

### 5.5.1 Definition

i) Ein Preflow $x$ ist eine Funktion $x : E \to \mathbb{N}_0$ mit

   a) $0 \leq x_{ij} \leq u_{ij}$ Kapazitätsbedingung

   b) $\sum_{\substack{\text{für ein-}\\ \text{gehende}\\ \text{Kanten}}} x_{ji} - \sum_{\substack{\text{für aus-}\\ \text{gehende}\\ \text{Kanten}}} x_{ik} \geq 0 \; \forall \; i \in V \setminus \{s,t\}$

   (dh. eventuell mehr einfließender als abfließender Fluss)

ii) $e(i) := \sum x_{ji} - \sum x_{ik}$ heißt Überschuss (oder Excess) von $i$.
Ein Preflow mit $e(i) = 0 \; \forall \; i \in V \setminus \{s,t\}$ ist ein Flow.

iii) Ein Knoten $i \in V \setminus \{s,t\}$ mit $e(i) > 0$ heißt aktiv.

**Idee für Algorithmus:**

- Schiebe (push) Fluss von $s$ nach $t$
  (von Knoten zu Knoten $\leftrightarrow$ erhöhender Pfad)
- Während des Ablaufs entsteht dadurch ein Preflow (mit Excess-Knoten)
- Am Ende soll der Preflow ein echter Flow sein.
  (dh. es existieren keine aktiven Knoten mehr)

Für die Richtung ($s \to t$) verwenden wir eine Distanzfunktion.

### 5.5.2 Distanz-Funktion

Die Distanz ist die Länge (= Anzahl) von Pfaden nach $t$.

## Definition

Für einen Preflow $x$ definieren wir in $G(x)$ eine Distanzfunktion $d = V \to \mathcal{N}_0$ mit

i) $d(t) = 0$

ii) $d(i) \leq d(j) + 1 \; \forall \; (i,j) \in G(x)$

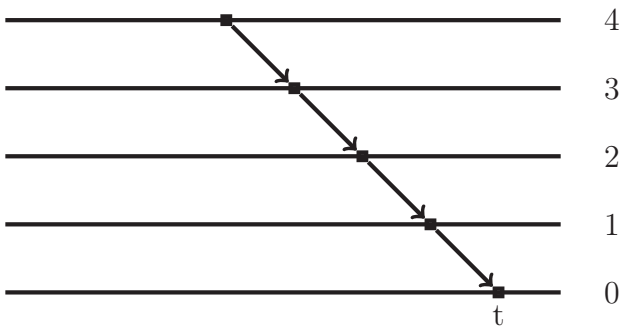
**Beobachtung:**

i) $d$ ist untere Schranke für exakte Distanzwerte.

ii) $d = 0$ (Nullfunktion) ist gültige Distanzfunktion.

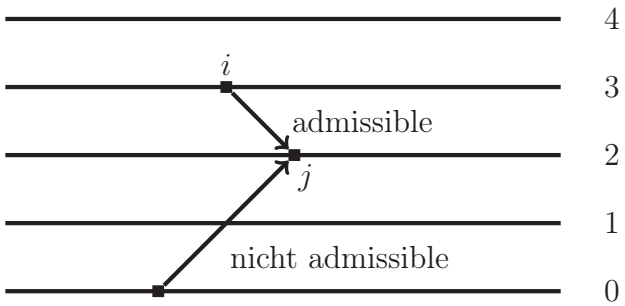
**Intention:**

$d(i)$ entspricht der Höhe oder dem Level des Knoten $i$.



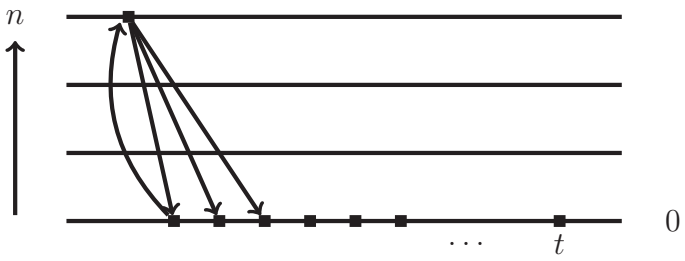

## Definition admissible

Eine Kante $(i,j)$ in $G(x)$ heißt admissible, wenn $d(i) = d(j) + 1$ dh. der Höhenunterschied genau 1 beträgt.



### 5.5.3 Preflow-Push-Algorithmus

1. Saturiere alle von $s$ ausgehenden Kanten.
2. Setze $d(i) = \begin{cases} 0, & \text{für } i \neq s \\ n, & \text{für } i = s \end{cases}$




$G(x)$

Überschussknoten = alle Nachbarn von $s$

3. Betrachte einen aktiven Knoten $i$ und schiebe Fluss über eine zulässige Kante
   $\rightarrow$ PUSH
4. Falls ein aktiver Knoten $i$ keine zulässige, ausgehende Kante hat
   $\rightarrow$ RELABEL
   $d(i) \leftarrow min\{d(j)|j \text{ Nachbar in } G(x)\} + 1$

### Der generische Preflow-Push-Algorithmus

Der generische Preflow-Push-Algorithmus nutzt keine besondere Strategie (wie z.B. highest-label, FIFO, ...) zur Auswahl aktiver Knoten.