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 δ.
Eine Flusserhöhung bzw. -verminderung um δ 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)

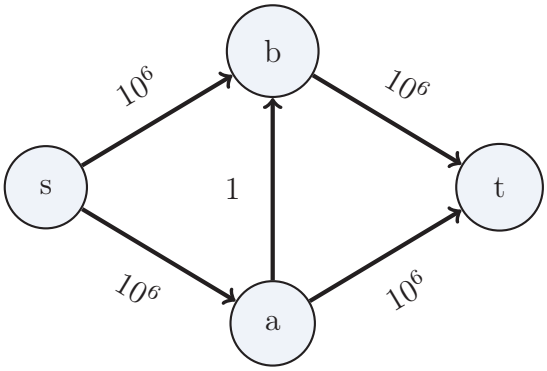
Beobachtung:
Höchstens $n-1$ Kanten verlaufen von s nach t

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

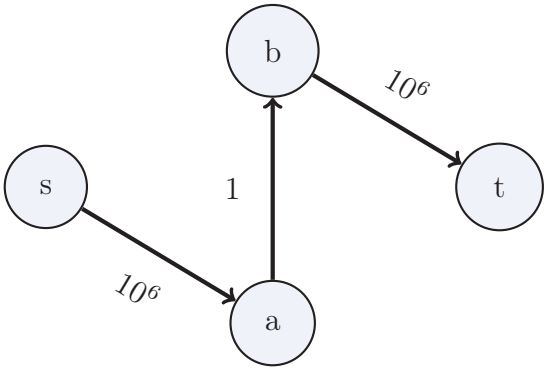
In jeder Iteration (AUGMENT) erhöht der Algorithmus den Flusswert um mindestens 1.
Daraus folgt, dass maximal F_{max} Iterationen benötigt werden, also maximal $n \cdot \mathcal{U}$ viele Iterationen.
Eine Iteration (Labeling) kostet Zeit $\mathcal{O}(n+m)$ (siehe DFS, BFS ...)
$\Rightarrow$ Gesamtlaufzeit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

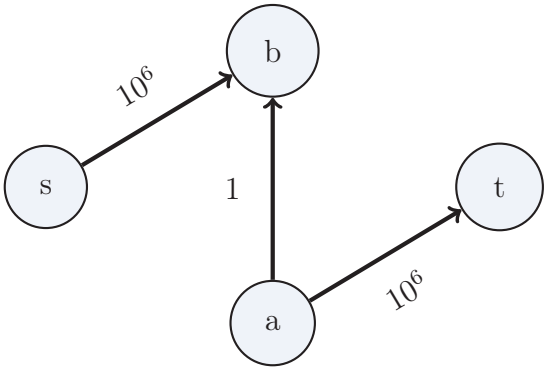
Annahme: G ist zusammenhängend.
$\Rightarrow m \geq n-1$
$\Rightarrow$ Laufzeit $\mathcal{O}(n \cdot m \cdot \mathcal{U})$
nicht polynomielle Laufzeit in m und n.

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 (δ) 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 Δ 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, $flow$, Δ)

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

Korrektheit

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

Laufzeit

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

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

Beweis
Betrachte das Ende einer Δ-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 Δ 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 Δ-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 Δ-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) $d(i) \leq d(j) + 1 \; \forall \; (i,j) \in G(x)$

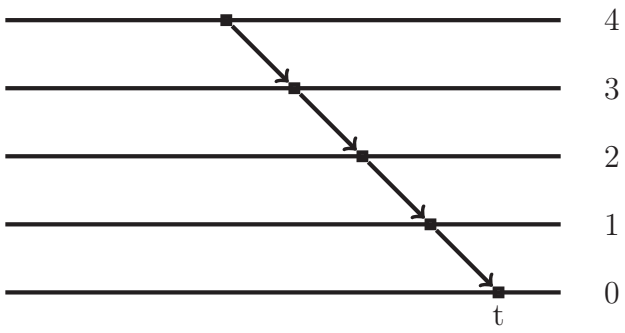
Beobachtung:

i) d ist untere Schranke für exakte Distanzwerte.

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

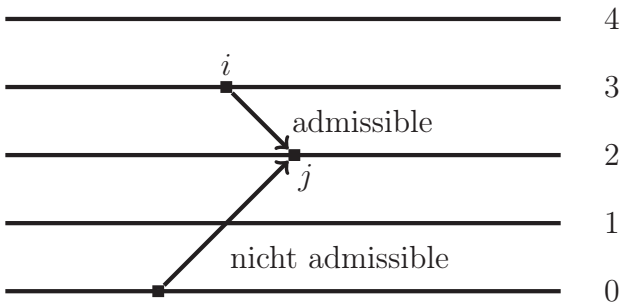
Intention:

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



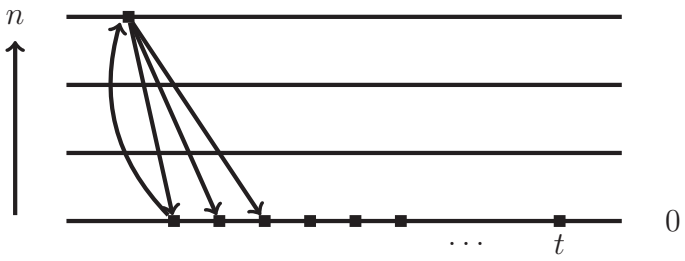
Definition admissible

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.

Algorithmus 4 : Generischer Preflow-Push-Algorithmus

foreach $v \in V$ **do**
  $d(v) \to 0$
  $e_v \to 0$
end
foreach $(u, v) \in E$ **do**
  $x_{uv} \to 0$
end
foreach $j \in V$ *mit* $(s, j) \in E$ **do**
  $x_{sj} \leftarrow u_{sj}$ // volle Kapazität
  $e_j \leftarrow e_j + x_{sj}$
end
$d(s) \leftarrow n$
while $\exists$ *aktive Knoten (dh.* $e(i) > 0$*)* **do**
  Wähle einen aktiven Knoten i
  PUSH/RELABEL(i)
end

Algorithmus 5 : Push/Relabel(i)

if $\exists$ *admissible Edge* (i, j) *in* $G(x)$ **then**
  Wähle eine solche Kante (i, j)
  $\delta \leftarrow min\{e(i), r_{ij}\}$ Überschuss, Restkapazität
  $x_{ij} \leftarrow x_{ij} + \delta$
  $e(i) \leftarrow e(i) - \delta$
  $e(j) \leftarrow e(j) + \delta$
end
else
  $d(i) \leftarrow min\{d(j)|(i, j)$ in $G(x)\} + 1$
end