5.4 Erhöhende-Pfad-Algorithmen

Der Labeling-Algorithmus ist ein Erhöhender-Pfad-Algorithmus (augmenting path).

5.4.1 Der allgemeine Labeling-Algorithmus

Algorithmus 3 : Allgemeiner Labeling-Algorithmus

```
1 x = 0 // Nullfunktion (alle x_ij = 0)
2 while G(x) enthält einen Pfad von s nach t do
3 |   Sei P ein solcher (erhöhender) Pfad
4 |   δ = min{r_ij|(i, j) ∈ P}
5 |   Erhöhe den Fluss x entlang von P um δ Einheiten
6 |   Berechne G(x) neu
7 end
```

5.4.2 Der Labeling-Algorithmus

(Konkrete Implementation des allgemeinen Algorithmus)

Idee:
(Ähnlich wie bei DFS, BFS, explore from ...)
Finde die Menge S aller Knoten, die von s aus in $G(x)$ erreichbar sind. Diese werden markiert. Wenn t markiert wurde, dann gilt $t \in S$, dh. es existiert ein erhöhender Pfad $\rightarrow$ Pfaderhöhung

Darstellung der Pfade:
Pred-Array (siehe kürzeste Wege)
$pred[v]$ ist die Kante über die v zum ersten Mal erreicht wurde.

```
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-

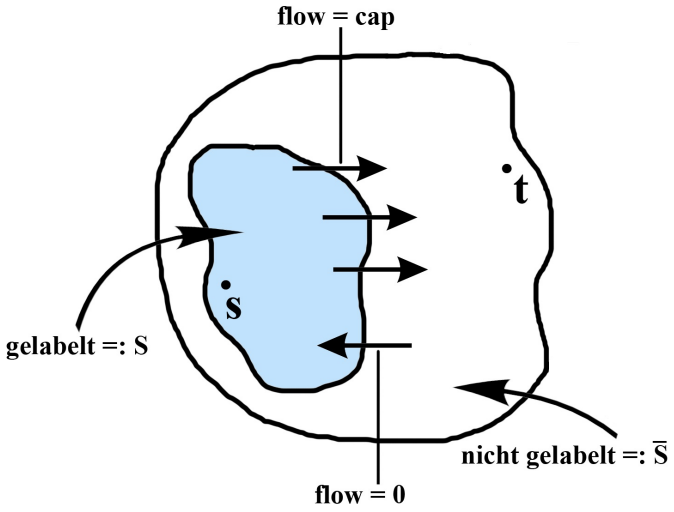
ge L hinzugefügt wird, wird s zu Beginn immer gelabelt und in L eingefügt. In der inneren Schleife wird zuerst ein beliebiger Knoten u aus der Menge L entnommen. Von diesem Knoten u aus werden nun sowohl die ausgehenden, als auch die eingehenden Kanten betrachtet. Dabei werden diejenigen Kanten, die 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 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)

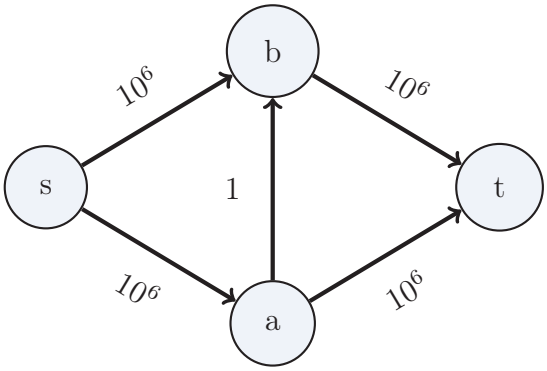
Beobachtung:
Höchstens $n-1$ Kanten verlaufen von s nach t

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

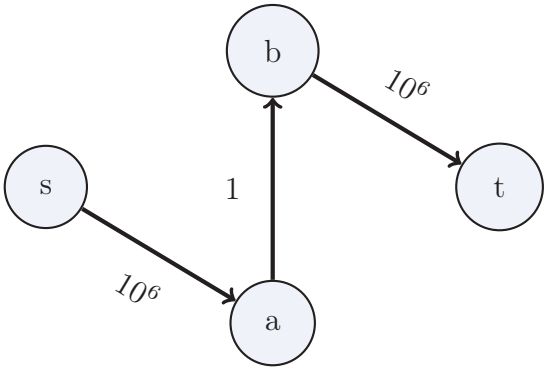
In jeder Iteration (AUGMENT) erhöht der Algorithmus den Flusswert um mindestens 1.
Daraus folgt, dass maximal F_{max} Iterationen benötigt werden, also maximal $n \cdot \mathcal{U}$ viele Iterationen.
Eine Iteration (Labeling) kostet Zeit $\mathcal{O}(n+m)$ (siehe DFS, BFS ...)
$\Rightarrow$ Gesamtlaufzeit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

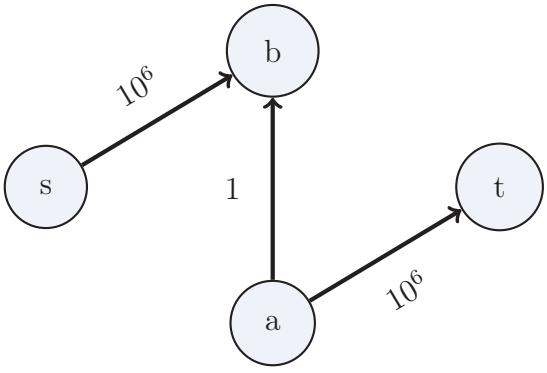
Annahme: G ist zusammenhängend.
$\Rightarrow m \geq n-1$
$\Rightarrow$ Laufzeit $\mathcal{O}(n \cdot m \cdot \mathcal{U})$
nicht polynomielle Laufzeit in m und n.

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).