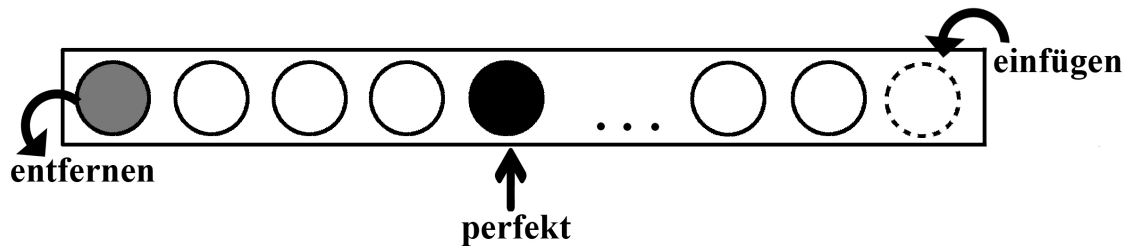


4.5.1 Allgemeiner Fall

Bei einem beliebigen Graph sind Zyklen möglich. Da auch die Kostenfunktion beliebig ist und somit negative Kosten auftreten können, können negative Zyklen entstehen.

Idee: Realisiere U als Schlage (FIFO)



Beinhaltet der Graph keine negativen Zyklen, so existiert nach Lemma 2 mindestens ein perfekter Knoten ($DIST[v] = dist(s, v)$) in der Schlange.

⇒ Zwischen zwei Entnahmen des selben Knoten u wurde mindestens ein perfekter Knoten entfernt. Dieser kann nicht wieder in die Schlange eingefügt werden.

⇒ Jeder Knoten wird maximal n -mal entfernt. Spätestens danach gilt $U = \emptyset$.

Beobachtung:

Wird ein Knoten öfter ($> n$) entfernt, dann muss nach Lemma 2 ein negativer Zyklus existieren.

→ Algorithmus von Bellman/Ford

Algorithmus von Bellman/Ford

```
1  bool BELLMANFORD(const graph& G, node s, const
   edge_array<int>& cost, node_array<int>& DIST,
   node_array<edge>& PRED){
2      queue<node> Q;
3      node_array<bool> inQ(G,false);
4      node_array<int> count(G,0);
5      node v;
6      forall_nodes(v,G){
7          DIST[v] = MAXINT;
8          PRED[v] = NULL;
9      }
10     DIST[s] = 0;
11     Q.append(s);
12     inQ[s] = true;
13     while(!Q.empty()){ // Solange Q nicht leer ist
14         node u = Q.pop();
15         inQ[u] = false;
16         if(++count[u] > G.numbers_of_nodes())
17             return false;
18         edge e;
19         forall_out_edges(e,u){
20             node v = G.target(e);
21             int d = DIST[u] + cost(e);
22             if(d < DIST[v]){
23                 DIST[v] = d;
24                 PRED[v] = e;
25                 if(!inQ[v]){
26                     Q.append(v);
27                     inQ[v] = true;
28                 }
29             }
30         }
31     }
32     return true;
33 }
```

Listing 4.1: Bellman/Ford

PRED-Verweise:

Die *PRED*-Verweise werden dazu genutzt, später den kürzesten Weg nachverfolgen zu können. *pred*[*v*] gibt dabei an, über welche Kante *v* auf dem kürzesten Weg erreicht wird. *PRED*-Verweise ändern sich daher, wenn der kürzeste Weg sich ändert.

Erklärung:

Zuerst wird *s* in die Menge *Q* aufgenommen, da sich dessen *dist*-Wert nicht mehr ändert, außer *s* liegt auf einem negativen Zyklus. Nun beginnt der Algorithmus richtig und läuft so lange, bis entweder *Q* leer ist und damit zu jedem Knoten ein kürzester Weg gefunden wurde oder der Algorithmus bricht ab, da er einen negativen Zyklus erkannt hat.

Innerhalb der *while*-Schleife wird ein beliebiger Knoten *u* aus *Q* gewählt und entfernt. Für alle ausgehenden Kanten von *u* wird die Δ -Ungleichung überprüft und im Fall, dass sie verletzt wird, wird der *DIST*-Wert des folgenden Knoten korrigiert, dessen *PRED*-Verweis auf *u* gesetzt und der Knoten in die Menge *Q* aufgenommen, da dessen ausgehenden Kanten die Δ -Ungleichung verletzen könnten.

Laufzeitanalyse:

$$n \cdot \left(\underbrace{\text{Iterationen über alle Knoten} + \text{ausgehende Kanten}}_{\mathcal{O}\left(\underbrace{\sum_{v \in V} (1 + \text{outdeg}(v))}_{n+m}\right)} \right)$$

Gesamtlaufzeit: $\mathcal{O}(n \cdot (n + m)) = \mathcal{O}(n^2 + n \cdot m)$

Wenn der Graph zusammenhängend ist, ist $m \geq n - 1$

$\Rightarrow$ Laufzeit: $\mathcal{O}(nm)$

dh. $\mathcal{O}(n^2)$ für dünne Graphen (Graphen mit wenig Kanten)

$\mathcal{O}(n^3)$ für dichte Graphen (Graphen mit vielen Kanten)

Ist der Graph nicht zusammenhängend, so kann die Zusammenhangskomponente, in der sich *s* befindet, in linearer Zeit gefunden werden.

4.5.2 Azyklische Graphen

In azyklischen Graphen treten keine Zyklen auf. Daher ist es auch, trotz beliebiger Kostenfunktion und dadurch möglicher negativer Kosten, nicht möglich, dass negative Zyklen auftreten. Dadurch existiert eine topologische Sortierung der Knoten.

**Lemma 3**

Der Knoten $u \in U$ mit kleinster topologischer Nummer, ist eine perfekte Wahl dh.
 $DIST[u] = dist(s, u)$

Beweis (indirekt)

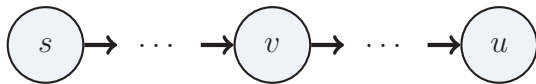
Wähle $u \in U$ mit $topnum[u]$ minimal.

Annahme: $DIST[u] > dist(s, u)$

Lemma 2 b) $\Rightarrow \exists$ Knoten v auf dem kürzesten Pfad von s nach u mit:

1) $DIST[v] = dist(s, v)$

2) $v \in U$



$\Rightarrow topnum[v] < topnum[u] \not\downarrow$

Widerspruch zur kleinsten Wahl von u (kleinste $topnum$ in U)

$\Rightarrow DIST[u] = dist(s, u)$

□

Mögliche Implementierungen:

- 1) Topsort $\rightarrow$ Liste aller Knoten in U mit aufsteigenden $topnum$'s
Durchlaufe diese Knoten
- 2) Kombiniertes Algorithmus aus Topsort und kürzeste Wege

Algorithmus Topsort und kürzeste Wege

```
1 void Topsort(const graph& G, node s, const edge_array<
  int> cost, node_array<int>& Dist){
2     node_array<int> INDEG(G,0);
3     forall_nodes(v,G){
4         INDEG[v] = indeg(v);
5         if(INDEG[v] == 0)
6             ZERO.append(v);
7         DIST[v] = MAXINT;
8     }
9     DIST[s] = 0;
10    while(!ZERO.empty()){
11        u = ZERO.pop();
12        edge e;
13        forall_out_edges(e,u){
14            v = G.target(e);
15            if(--INDEG[v] == 0)
16                ZERO.append(v);
17            int d = DIST[u] + cost(u,v);
18            if(d < DIST[v])
19                DIST[v] = d;
20        }
21    }
22 }
```

Listing 4.2: Topsort und kürzeste Wege

Erklärung:

Die Grundstruktur entspricht dem normalen Topsort-Algorithmus (ZERO-Liste). In die innere Schleife wurde die Überprüfung der Δ -Ungleichung eingebaut. Da dieser zusätzliche Aufwand in konstanter Zeit ausführbar ist, ändert sich die Laufzeit von Topsort nicht.

Der Knoten s muss hierbei nicht gesondert behandelt werden, da der Algorithmus von alleine erst richtig startet, wenn s betrachtet wird, da die *dist*-Werte aller anderen Knoten auf *MAXINT* gesetzt wurden und somit die Δ -Ungleichung nie verletzt wird.

Laufzeit:

$\mathcal{O}(n + m)$ (perfekte Wahl!)

4.5.3 Nicht-negative Netzwerke

In nicht-negativen Netzwerken können Zyklen vorkommen, aber aufgrund der fehlenden negativen Kosten, können keine negativen Zyklen entstehen.

Idee:

Wähle in Zeile 7 des Algorithmus 2 einen Knoten $u \in U$ mit $DIST[u]$ minimal. Dafür eignet sich eine Priority Queue und deren Funktion $PQ.delmin()$.

→ Dijkstra

Lemma 4

Der Knoten $u \in U$ mit $DIST[u]$ minimal, dh. $DIST[u] = dist(s, u)$, ist eine perfekte Wahl.

Beweis (indirekt)

Annahme: $DIST[u] > dist(s, u) \xrightarrow{\text{Lemma 2 b)}} \exists v \in U$ mit $DIST[v] = dist(s, v)$ auf einem kürzesten Pfad von s nach u ($v \neq u$).

Dann gilt:

$$\begin{array}{ll} dist[s, u] \geq dist(s, v) & \text{da alle Kosten nicht-negativ sind} \\ = DIST[v] & \text{nach Lemma 2 b)} \\ \geq DIST[u] & \text{da } u \text{ minimal gewählt wurde} \end{array}$$

$\Rightarrow dist(s, u) = DIST[u]$ da $DIST$ -Werte nie zu klein werden. $\nmid$ Widerspruch

Daher gilt: $DIST[u] = dist(s, u)$

□

Implementierung

Datenstruktur: Priority Queue

LEDA-Datentyp:

- i) allgemein: $p_queue < I, P >$
 I = Information, P = Priorität

hier: $I = \text{node}$
 $P = \text{int DIST-Werte}$

- ii) Für Graphen: $\text{node_pq} < P > PQ$ (Knoten-Priority-Queue)
 $PQ = \text{Menge von Knoten mit dazugehörigen Prioritäten}$

Operationen:

- Konstruktor: $\text{node_pq} < P > PQ(\text{graph } G)$
→ leere PQ für Knoten von G
- void $PQ.\text{insert}(\text{node } v, P p)$ fügt einen Knoten v mit der Priorität p zu PQ hinzu.
- node $PQ.\text{delmin}()$ entfernt einen Knoten v mit kleinster Priorität und gibt v zurück.
- node $PQ.\text{find_min}()$ gibt einen Knoten v mit kleinster Priorität zurück.
- void $PQ.\text{decrease_p}(\text{node } v, P q)$ vermindert die Priorität von v auf q .
- bool $PQ.\text{empty}()$ testet, ob PQ leer ist.

Algorithmus von Dijkstra

```
1 void Dijkstra(const graph& G, node s, const edge_array<
  int>& cost, node_array<int>& DIST, node_array<edge>&
  PRED){
2     node_pq<int> PQ(G);
3     node v;
4     forall_nodes(v,G){
5         DIST[v] = MAXINT;
6         PRED[v] = NULL;
7     }
8     DIST[s] = 0;
9     PQ.insert(s,0);
10    while(!PQ.empty()){
11        node u = PQ.del_min(); // perfekte Wahl
12        egde e;
13        forall_out_edges(e,u){
14            node v = G.target(e);
15            int d = DIST[u] + cost[e];
16            if(d < DIST[v]){
17                if(DIST[v] == MAXINT)
18                    PQ.insert(v,d);
19                else // v ist in U
20                    PQ.decrease_p(v,d)
21                DIST[v] = d;
22                PRED[v] = e;
23            }
24        }
25    }
26 }
```

Listing 4.3: Dijkstra

Erklärung:

Zuerst wird s in die Priority Queue PQ aufgenommen. Die *while*-Schleife läuft nun so lange, bis PQ leer ist. Ist dies der Fall, ist zu jedem Knoten ein kürzester Weg bekannt. In der *while*-Schleife wird zuerst aus der PQ ein Knoten u gewählt. Dabei wird darauf geachtet, dass es eine perfekte Wahl ist. Dann werden alle von u ausgehenden Kanten betrachtet, die jeweilige Δ -Ungleichung überprüft und falls sie verletzt wurde, der erreichte Knoten entweder in PQ aufgenommen, falls er noch nicht in PQ drin ist oder sonst seine Priorität auf seinen neuen $DIST$ -Wert gesetzt.

Danach wird noch sein *DIST*-Wert angepasst und sein *PRED*-Verweis auf die Kante gesetzt, über die er erreicht wurde.

Laufzeitanalyse:

i) Operationen auf dem Graphen: $\mathcal{O}(n + m)$

ii) Priority Queue:

1 · Konstruktor

$n \cdot \text{insert}()$

$n \cdot \text{delmin}()$

$n \cdot \text{empty}()$

$m \cdot \text{decrease}()$

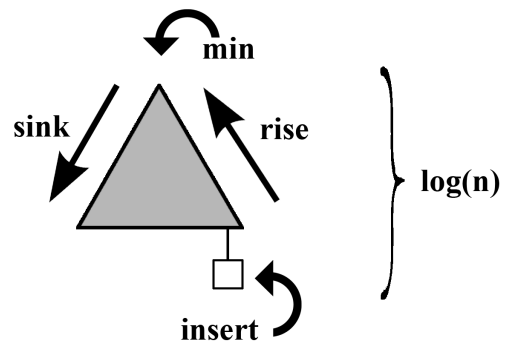
$\Rightarrow$ Gesamtlaufzeit: $\mathcal{O}(n \cdot (T_{\text{insert}}(n) + T_{\text{delmin}}(n) + T_{\text{empty}}(n)) + m \cdot T_{\text{decrease}}(n))$

$T_{\text{op}}(n)$ = Laufzeit der Operation *op* auf PQ der Größe *n*

Varianten von Datenstrukturen:

- binärer Min-Heap
- balancierter Baum

$\Rightarrow$ Dijkstra $\mathcal{O}((n + m) \cdot \log(n))$



Spezielle Heap-Datenstruktur:

Ein Fibonacci-Heap ist eine Menge von binomischen Bäumen. Daher ist der maximale Rang eines Fibonacci-Heaps kleiner gleich $\log(n)$, wobei *n* die Anzahl der Knoten ist. Zudem besitzt der Fibonacci-Heap einen Pointer auf eine Wurzel mit minimaler Priorität. Hier entspricht die Priorität den *DIST*-Werten. Dadurch ergibt sich der folgende Aufwand:

Amortisierte Analyse:

$$\left. \begin{array}{l} \text{Insert } \mathcal{O}(1) \\ \text{Delmin } \mathcal{O}(\log(n)) \\ \text{Decrease } \mathcal{O}(1) \end{array} \right\} \rightarrow \mathcal{O}(n \cdot \log(n) + m)$$

5 Maximale Flüsse

5.1 Transport-Problem

Ein Unternehmer möchte täglich möglichst viele seiner Waren von einem seiner Standorte zu einem anderen Standort transportieren. Er nutzt dazu LKW's anderer Unternehmen, die zwischen verschiedenen Städten hin und her fahren. Jeder hat eine maximale Kapazität, die er von einer zu einer andern Stadt transportieren kann. Jeder dieser LKW's hat allerdings eine maximale Kapazität an Waren, die er transportieren kann. Da es in den einzelnen Städten keine Lagerhäuser gibt, können die Waren nicht zwischengelagert werden, sondern müssen sofort weiter transportiert werden.

Der Ausgangsstandort in diesem Szenario stellt den Startknoten s und der zweite Standort den Endknoten t dar. Die LKW 's fahren entlang der Kanten und ihre Kapazität entspricht der Kapazität der jeweiligen Kante.

Frage: Was ist der maximale Fluss von Waren von s nach t ?

5.1.1 Formale Definition

Angelehnt an das Buch „Network Flows“ von Ahuja, Magnanti und Orlin.

Gegeben sei

- ein gerichteter Graph $G = (V, E)$
- eine Kapazitätsfunktion $u : E \rightarrow \mathbb{R}_0^+$ und
- zwei Knoten $s, t \in V$ mit $s \neq t$.

$u((v, w))$ heißt Kapazität von (v, w) . s heißt Quelle (source) und t Senke (target).

Gesucht ist eine Flussfunktion $x : E \rightarrow \mathbb{R}_0^+$ mit folgenden Eigenschaften:

- i) Kapazitätsbedingung:
 $\forall e \in E : 0 \leq x(e) \leq u(e)$

ii) Massenbalance-Bedingung:

Sei $v \in V$ (beliebig aber fest!)

$$\underbrace{\sum_{(v,u) \in E} x(v,u)}_{\substack{\text{gehe über alle} \\ \text{ausgehenden} \\ \text{Kanten von } v}} - \underbrace{\sum_{(w,v) \in E} x(w,v)}_{\substack{\text{gehe über alle} \\ \text{eingehenden} \\ \text{Kanten von } v}} = \begin{cases} F, & v = s \\ 0, & \forall v \in s, t \\ -F, & v = t \end{cases}$$

$$F \geq 0$$

ii) Optimalitätsbedingung:

F soll maximal sein.

Erklärungen:

- i) Für alle Kanten muss gelten, dass ihr Flusswert zwischen Null und ihrer maximalen Kapazität liegt.
- ii) Für alle Knoten die nicht s oder t sind, muss gelten, dass alles was in sie rein fließt auch wieder abfließt. Es kann also nichts in diesen Knoten gelagert werden. F entspricht dem Fluss der fließt. Aus dem Knoten s fließt der komplette Fluss F heraus, aber nichts herein. Beim Knoten t ist es genau umgekehrt.
- iii) Ohne diese Bedingung wäre das Problem trivial, da der Null-Fluss ($x = 0$) die anderen beiden Bedingungen immer erfüllt.

Notation:

Knoten: i, j

Die Kapazität wird anstatt mit $u((i, j))$ jetzt nur noch mit u_{ij} und die Flussfunktion anstatt mit $x((i, j))$ mit x_{ij} bezeichnet.

5.1.2 Das Restnetzwerk

(residual network)

Ein Restnetzwerk beschreibt mögliche Flussrichtungen.

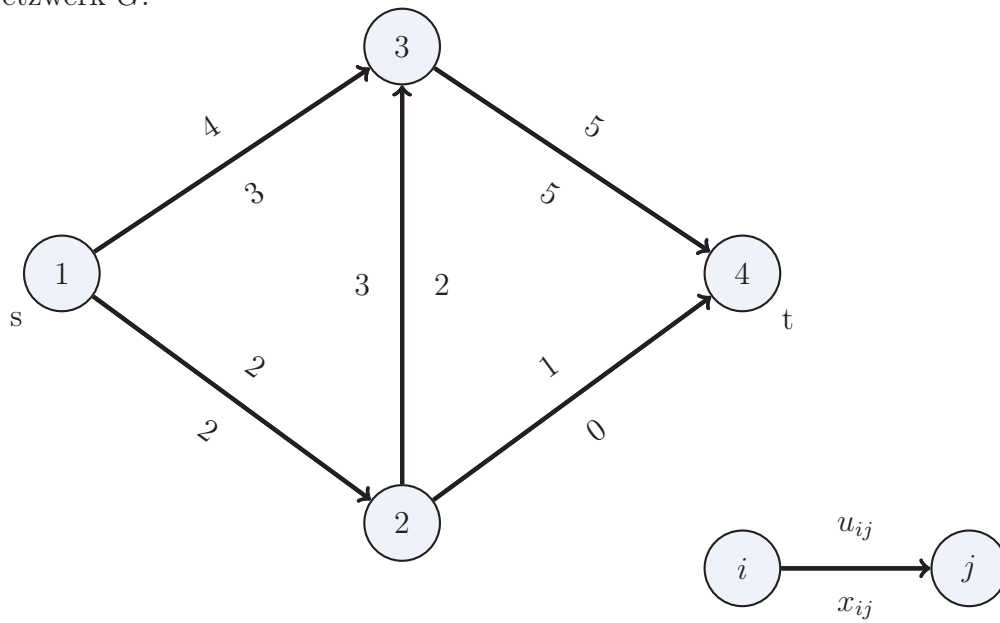
Definition

Sei x eine aktuelle Flussfunktion, dh. sie erfüllt die Kapazitätsbedingung (z.B. der Nullfluss ($\forall i, j \ x_{ij} = 0$)).

Das Restnetzwerk $G(x)$ besteht aus allen Kanten von G und deren Gegenkante. Es existieren also für jede Kante (i, j) aus G zwei Kanten in $G(x)$, nämlich die Kante (i, j) und ihre Gegenkante (j, i) . Sie besitzen die Restkapazitäten $r_{ij} = u_{ij} - x_{ij}$ und $r_{ji} = x_{ij}$.



Die Restkapazitäten beschreiben mögliche Änderungen an der Flussfunktion. Eine Erhöhung geschieht in Richtung der Kante (i, j) ($\max r_{ij} = u_{ij} - x_{ij}$) und eine Verminderung in die Gegenrichtung ($\max r_{ji} = x_{ij}$).

**Beispiel 5.1**Netzwerk G :Aktueller Flusswert: $F = 5$ Restnetzwerk $G(x)$: