

Netzwerkalgorithmen

July 4, 2016

1 Zusätzliches blabla

Makros in C/C++: `#define alias replace`, wobei `replace` auch Code sein kann.

2 Datentypen für Graphen und Netzwerke (LEDA)

Definition eines Datentyps

Definition der Objekte des Typs: `stack < T >`

Konstruktion: `stack < int > S(100)` (max Größe)

Operationen: `s.push(Tx)`, `ts.pop()`

Bemerkung zu Implementierung

Graph-Datentyp in LEDA

Der Typ `graph` repräsentiert gerichtete Graphen.

Ein Graph `g` besteht aus zwei Typen von Objekten: `node` und `edge`

Mit jedem Knoten `v` sind zwei Listen von Kanten (`list < edge >`) verbunden (eingehend und ausgehend)

Mit jeder Kante `e` werden 2 Knoten `source` und `target` gespeichert.

Operationen auf G

Update:

`node G.new_node()`, erzeugt einen neuen Knoten in `G` und gibt ihn zurück. `edge G.new_edge(node v, node w)`

`void G.del_edge(edge)`

Access:

`list < edge > G.out_edges(node v);`

`int G.outdeg(node v);`

`node G.source(edge);`

`node G.target(edge);`

Iteration:

`forall_nodes(v,G)`

`forall_edges(e,G)`

`forall_out_edges(e,v)`

`forall_in_edges(e,v)`

1. Problem

Gegeben: Graph $G=(V,E)$

Frage: Ist G azyklisch?

Algorithmus siehe Topologisches Sortieren: Entferne jeweils einen Knoten v mit $\text{indeg}(v)=0$ bis der Graph leer ist. Falls wir keinen solchen Knoten finden dann ist der Graph zyklisch, falls G am Ende leer, ist er azyklisch.
C++:

```
bool ACYCLIC(graph G){                               //Call by value damit G nicht zerstört
    list<node> zero;
    node v;
    forall_nodes(v,G){
        if (G.indeg(v)==0) zero.append(v);
    }
    while (!zero.empty()){
        node u = zero.pop();
        edge e;
        forall_out_edges(e,u){
            node w=G.target(e);
            G.del_edge(e);
            if (G.indeg(w) == 0){
                zero.append(w);
            }
        }
    }
    return G.empty();
}
```

Daten für Knoten und Kanten

1. Parametrisierte Graphen: $\text{GRAPH}<\text{node_type}, \text{edge_type}> G$
2. Temporäre Daten: $\text{besucht}[v] \leftarrow \text{true}$

Datentypen in LEDA

$\text{node_array}<T> A(G,x)$: Feld über die Knoten des Graphen G $\text{edge_array}<T> B(G,y)$ analog Verwendet für: Temporäre Daten, Eingabedaten, Resultate

Anwendung im topologischen Sortieren

injektive Abbildung: $\text{topnum}: V \rightarrow \{1, \dots, n\}$ mit $\forall (v, w) \in E: \text{topnum}[v] < \text{topnum}[w]$

```
bool TOPSORT(const graph& G, node_array<int>& topnum){
    int count = 0;
    list<node> zero;
    node_array<int> indeg(G);
    node v;
    forall_nodes(v,G){
        indeg[v] = G.indeg(v);
        if (indeg[v] == 0) zero.append(v);
    }
    while (!zero.empty()){
        node v = zero.pop();
        topnum[v] = ++count;
        edge e;
        forall_out_edges(e,v){
            node w = G.target(e);
            if (--indeg[w] == 0) zero.append(w);
        }
    }
}
```

```

    }
}
return count == G.number_of_nodes();
}

```

Tiefensuche

Hauptprogramm:

```

void DFS(const graph& G, node_array<int>& dfsnum, node_array<int>& compnum){
    int count1 = 0;
    int count2 = 0;
    node_array<bool> visited(G, false);
    node v;
    forall_nodes(v,G){
        if (!visited[v]) dfs(G,v,count1,count2,dfsnum,compnum)
    }
}

```

Rekursive Funktion dfs:

```

void dfs(const graph& g, node v, int& count1, int& count2, node_array<int>& dfsnum, -
    node_array<int>& compnum){
    dfsnum[v] = ++count1;
    visited[v] = true;
    edge e;
    forall_out_edges(e,v){
        edge w = G.target(e);
        if (!visited[w]) dfs(G,w,count1,count2,dfsnum,compnum)
    }
    compnum[v] = ++count2
}

```

Berechnung starker Zusammenhangskomponenten

Definition: Ein gerichteter Graph ist stark zusammenhängend, wenn $\forall v, w \in V : v \rightarrow^* w$ (es existiert ein Pfad von v nach w)

Die starken Zusammenhangskomponenten (SZK) von G sind die maximalen SZK Teilgraphen von G .

Idee für Algorithmus:

1. führe DFS mit $G' = (V', E')$ dem Teilgraphen aufgespannt von bereits besuchten Knoten
2. Verwalte SZK von G' während DFS ausgeführt wird.

Ablauf:

Sei (v,w) die nächste in dfs betrachtete Kante

1. Fall: $(v,w) \in T$ (Baumkante), w noch nicht besucht. $V' = V' \cup \{w\}, E' = E' \cup \{(v,w)\}, SZK = SZK \cup \{\{w\}\}$
2. Fall: $(v,w) \notin T$, d.h. w wurde schon besucht, ist also in V' enthalten. $E' = E' \cup \{(v,w)\}$. Nun kann (v,w) mehrere bereits bekannte SZK vereinigen (Rückwärtskante oder Cross-Kante). Bemerkung: Vorwärtskanten generieren keine neuen Pfade in G' , deswegen dabei keine Änderung der SZK.

Bezeichnungen:

1. Eine SKZ K heißt abgeschlossen, falls die Aufrufe von dfs für alle Knoten v in K beendet sind.
2. Die Wurzel einer SZK K ist der Knoten mit der kleinsten dfsnum in K .
3. "Unfertig" ist die Folge aller Knoten für die dfs aufgerufen wurde, aber deren SZK in der sie sich befinden noch nicht abgeschlossen ist.

4. "Wurzeln" ist die Folge aller Wurzeln der nicht abgeschlossenen SZK nach dfsnum sortiert.

Situation, wenn DFS beim Knoten g angekommen ist:

Unfertig: a, b, c, e, f, g

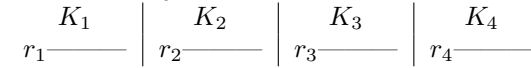
Wurzeln: $1, b, e, g$

Der Algorithmus betrachtet danach die Kanten aus g (g, d) $\in C$: es passiert nichts, da d in einer abgeschlossenen SZK ist; (g, c) $\in C$ Vereinigt die 3 SZK mit den Wurzeln b, e, g durch entfernen von e, g aus der Wurzelfolge.

Beobachtung: Hinzufügen und Streichen nur am Ende \rightarrow Stack eignet sich als Datenstruktur.

Allgemeine Situation für $(v, w) \in T$:

$K' = K_2 \cup K_3 \cup K_4$.



Ergänzungen von DFS für SZK:

1. Aktion: while $\text{dfsnum}[\text{wurzel.top}()] > \text{dfsnum}[w]$ do $\text{wurzel.pop}()$ od
2. Falls $(v, w) \in T$: $\text{Wurzeln.push}(w)$; $\text{Unfertig.push}(v)$
3. Abschluss eine SZK: SZK von v wird endgültig verlassen, sie ist nun abgeschlossen.

Am Ende von $\text{dfs}(v)$: if $v == \text{Wurzel.top}()$ then

$\text{Wurzeln.pop}()$;

repeat $w = \text{unfertig.pop}()$ until $w == v$

fi

Übung 2: Algo zu SZK.

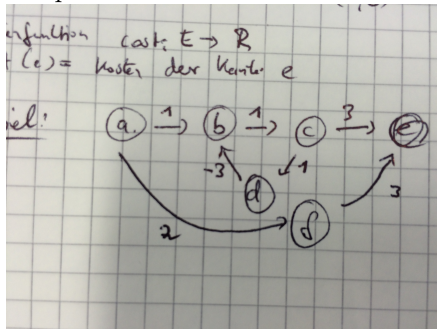
Darstellung der einzelnen SZKs: $\text{node_array} < \text{int} > \text{szknum}(G)$; $k = \# \text{kompnum}$ forall $v \in V$ $\text{szknum}[v] = i \Leftrightarrow v$ in der Komponente i

$\rightarrow \text{int SZK}(\text{const graph\& } G, \text{node_array} < \text{int} > \& \text{sznum})$

3 Kürzeste Wege / Billigste Wege

Allgemeines Problem: Sei gegeben ein gerichteter Graph $G = (V, E)$, eine Kostenfunktion $\text{cost} : E \rightarrow \mathbb{R}$ mit $\text{cost}(e) = \text{Kosten der Kante } e$.

Beispiel:



$k=3$

Kosten eines Pfades P : $\text{cost}(P) := \sum_{e \in P} \text{cost}(e)$

Billigste-Wege Problem: Finde einen Pfad P mit $\text{cost}(P)$ minimal

- a) Zwischen zwei gegebenen Knoten
- b) Von einem Knoten s zu allen anderen Knoten (single source shortest paths)
- c) Zwischen allen Paaren $(v, w) \in V \times V$

Im Beispiel: $\text{dist}(a, f) = 2$, $\text{dist}(a, e) = -\infty$ weil $a \rightarrow b \rightarrow (c \rightarrow d \rightarrow b \rightarrow c)^i \rightarrow e$

Definition: $\text{dist}(v, w) := \inf\{\text{cost}(P) \mid P \text{ ist Pfad von } v \text{ nach } w\}$ d.h. wenn es einen negativen Zyklus gibt $= -\infty$, wenn kein Pfad existiert: ∞ .

Wir betrachten hier die Single-Source Variante:

Eingabe: $G = (V, E)$, $s \in V$, $\text{cost} : E \Rightarrow \mathbb{R}$

Ausgabe: $\forall v \in V : \text{dist}(s, v)$

Beobachtung: Die dist -Funktion erfüllt die Dreiecksungleichung: $\text{dist}(s, w) \leq \text{dist}(s, v) + \text{cost}(v, w)$

Idee für allgemeinen Algorithmus (Label Correcting):

Überschätze die dist-Werte ($DIST[v] = \infty$). Solange eine Kante (u,v) die Dreiecksungleichung verletzt, korrigiere $DIST[v]$

```

for all v in V do
     $DIST[v] = \infty$ 
od
 $DIST[s] = 0$ 
while  $\exists (u,v) \in E$  mit  $DIST[v] > DIST[u] + cost(u,v)$ 
     $DIST[v] = DIST[u] + cost(u,v)$ 
od

```

Eigenschaften

- Es gilt immer $DIST[v] \geq dist(s,v)$
- Wenn $DIST[v] < \infty$, dann existiert ein Pfad von s nach v mit den Kosten $DIST[v]$
- Die kürzesten Pfade bilden einen Baum T mit Wurzel s. Begründung: In jeden Knoten mit $DIST[v] < \infty$ mündet genau ein kürzester Pfad. Bei korrigieren von DIST-Wert wird die eingehende Kante von T definiert.
- Für jede Kante (v,w) auf einem Kürzesten Pfad gilt: $dist(s,w) = dist(s,v) + cost(v,w)$

Da $DIST[v] < \infty$ existiert ein Pfad von s nach v mit $cost(P) = DIST[v]$. Daraus folgt $DIST[v] \geq dist(s,v)$, da $dist(s,v) = \inf\{cost(P) \mid P \text{ ist Pfad von s nach v}\}$

Effizienter Algorithmus: Kandidatenmenge

Wir speichern alle Knoten aus denen Kanten ausgehen können, die die Dreiecksungleichung verletzen in einer Menge U.

Am Anfang gilt: $U = \{s\}$. Immer wenn $DIST[v]$ vermindert wird, nimm v nach U auf. Verfeinerter Algorithmus:

```

1. for all v in V do
     $DIST[v] = \infty$ 
od
2.  $DIST[s] = 0$ 
3.  $U = \{s\}$ 
4. while  $U \neq \emptyset$ 
5.     wähle und entferne ein  $u \in U$ 
6.     for all  $v \in V$  mit  $(u,v) \in E$  do
7.          $c = DIST[u] + cost(u,v)$ 
8.         if  $c < DIST[v]$  then
9.              $DIST[v] = c$ 
10.             $U = U \cup \{v\}$ 
11.        fi
12.    od
13. od

```

Eigenschaften des Algorithmus während Laufzeit (Lemma):

Falls der Graph G keine negativen Zyklen enthält, dann gilt:

- Falls $v \notin U$, dann gilt für alle ausgehenden Kanten (v,w) : $DIST[w] \leq DIST[v] + c(v,w)$.
- Sei v_0, \dots, v_k ein billigster Pfad von s nach v. Falls nun $DIST[v] > dist(s,v)$, dann existiert ein i $0 \leq i \leq k-1$ mit $DIST[v_i] = dist(s,v_i)$ und $v_i \in U$
- Es existiert immer ein $u \in U$ mit $DIST[u] = dist(s,u)$ und wenn in Zeile 5 des Algorithmus ein solches u gewählt wird (perfekte Wahl), dann wird die while-Schleife für jeden Knoten höchstes einmal ausgeführt.

Beweis:

- Induktion über Schleifendurchläufe i (while)

i=0: Vor dem ersten Durchlauf gilt die Behauptung, da $DIST[s]=0$ und $DIST[v]=\infty \forall v \in V$, $U=\{s\}$

i=i+1: Betrachte ein beliebiges $v \notin U$ nach der i+1-ten Ausführung

$v \notin U$ vor der (i+1)-ten Ausführung -> Nach IA: $DIST[v] + c(v,w) \geq DIST[w]$ für alle ausgehenden Kanten

(v,w) und $DIST[v]$ wurde in diesem Durchlauf nicht verändert. Die $DIST$ -Werte der Nachbarn w von v wurden eventuell verändert $\Rightarrow DIST[v] + c(v,w) \geq DIST[w]$ für alle ausgehenden Kanten

$v \in U$ vor der $(i+1)$ -ten Iteration: v wurde in Zeile 5 ausgewählt, dann stellt die innere Schleife die Dreiecksungleichung für alle ausgehenden Kanten wieder her.

b) Sei $s = v_0, \dots, v_k = v$ ein billigster Pfad von s nach v mit $DIST[v_k] > dist(s, v_k)$

Sei i maximal mit $DIST[v_i] = dist(s, v_i) \Leftarrow 0 \leq i < k$ (v_0 hat korrekten Wert)

Wir zeigen $v_i \in U$:

Annahme $v_i \notin U$: $DIST[v_i] + c(v_i, v_{i+1}) \geq DIST[v_{i+1}]$ Außerdem gilt: $dist(s, v_{i+1}) = dist(s, v_i) + c(v_i, v_{i+1})$. Daraus folgt: $dist(s, v_{i+1}) = dist(s, v_i) + c(v_i, v_{i+1}) = DIST[v_i] + c(v_i, v_{i+1}) \geq DIST[v_{i+1}] \Rightarrow dist(s, v_{i+1}) = DIST[v_{i+1}]$ Widerspruch zur Wahl von i .

c) Sei $v \in U$ beliebig

Falls $DIST[u] = dist(s,u)$ ok.

Falls $DIST[u] > dist(s,u)$: Betrachte einen billigsten Pfad von s nach u , auf dem nach b) ein $v_i \in U$ existiert, mit $DIST[v_i] = dist(s, v_i)$

Perfekte Wahl: Falls in Zeile 5 immer ein solcher Knoten $u \in U$ gewählt wird, dann kann u kein zweites Mal nach U angefügt werden.

Perfekte Wahl \Rightarrow Gesamtaufwand: $\mathcal{O}(\sum_{v \in V} (1 + outdeg(v) + Verwaltung\ der\ Menge\ U)) = n + m + ?$

1. Allgemeiner Fall: beliebiger Graph mit beliebiger cost-funktion: Realisiere U als Schlange (FIFO)

Keine Negativen Zyklen \Rightarrow in der Schlange existiert mindestens ein perfekter Knoten \Rightarrow Zwischen 2 aufeinanderfolgenden Entnahmen des selben Knoten u wurde mindestens ein perfekter Knoten entfernt. \Rightarrow Jeder Knoten wird maximal n -mal entfernt (spätestens dann ist U leer).

Beobachtung: Wird ein Knoten öfters als n -mal entfernt, existiert ein negativer Zyklus

Bellman/Ford Algorithmus:

```
bool BELLMAN(const graph& G, node s, const edge_array<int>& cost, node_array<int>& DIST){
    queue<node> Q; //Menge U
    node_array<bool> inQ(G, false);
    node_array<int> count;
    DIST[s]=0;
    Q.append(s);
    inQ[s] = true;
    while (!Q.empty()){
        node u = Q.pop();
        inQ[u] = false;
        if (++count[u] > G.number_of_nodes()){
            return false;
        }
        edge e;
        forall_out_edges(e,u){
            node v=G.target();
            int c = DIST[u]+cost[e];
            if (c<DIST[v]){
                DIST[v] = c;
                //PRED würde hier gesetzt
                if (!inQ[v]){
                    Q.append(v);
                    inQ[v]=true;
                }
            }
        }
    }
    return true;
}
```

Laufzeitanalyse: $n \cdot (\text{Iterationen über alle Knoten} + \text{deren ausgehenden Kanten}) = \mathcal{O}(\sum_{v \in V} (1 + outdeg(v)))$

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

Wenn G zusammenhängend gilt $m \geq n - 1 \Rightarrow \mathcal{O}(n \cdot m)$

Problem: Falls negativer Zyklus existiert, gebe einen oder alle aus. 2. Azyklische Graphen: Es existiert eine topologische Sortierung.

Behauptung: Der Knoten $u \in U$ mit kleinster topologischer Nummer ist eine perfekte Wahl mit $\text{DIST}[u] = \text{dist}(s,u)$

Beweis: Sei $u \in U$ mit $\text{topnum}[u]$ minimal

Annahme: $\text{DIST}[u] > \text{dist}(s,u) \Rightarrow \exists$ Knoten v auf dem kürzesten Pfad von s nach u mit $\text{DIST}[v] = \text{dist}(s,u)$ und $v \in U$
 $\Rightarrow \text{topnum}[v] < \text{topnum}[u]$ Widerspruch.

3. Nicht-Negative Netzwerke: Beliebiger Graph aber Kosten ≥ 0

Dijkstra...

Wähle in Zeile 5 einen Knoten $u \in U$ mit $\text{DIST}[u]$ minimal. \Rightarrow Priority-Queue.

Behauptung: $u \in U$ mit $\text{DIST}[u]$ minimal ist perfekt d.h. $\text{DIST}[u] = \text{dist}(s,u)$

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

Dann gilt: $\text{dist}(s,u) \geq_{\text{nichtNegativ}} \text{dist}(s,u) =_{\text{Lemma}} \text{DIST}[v] \geq_{\text{minimalwahl}} \text{DIST}[u]$
 $\Rightarrow \text{dist}(s,u) = \text{DIST}[u]$ da DIST-Label nie zu klein werden.

Darstellung der billigsten Pfade durch pred-Verweise: Merke für jeden Knoten die eingehende Kante des kürzesten Weges.

Def: `node_array<edge> PRED`

Init: `forall_nodes(v,G) PRED[v] = NULL`

in der inneren Schleife (`forall_out_Edges(e,u): if(c<DIST[v]){ DIST[v] = c; PRED[v] = e }`)

Durchlaufen der Pfade:

```
while(PRED[v] != NULL){
    edge e=PRED[v]
    print e
    v=G.source(e)
}
```

Dijkstra-Algorithmus: Wähle Knoten u mit $\text{DIST}[u]$ minimal.

Datenstruktur: Priority Queue

LEDA: `node_pq < P > PQ` Knoten-Priority-Queue.

Operationen:

- Konstruktor `node_pq < P > PQ(Graph G)` Mit leerer PQ für Knoten von G
- `void insert (node v, P p)`
- `node PQ.del_min()` entfernt einen Knoten v mit kleinster Priorität und gibt diesen zurück
- `node PQ.find_min()`
- `void PQ.decrease_p(node v P p)` vermindert die Priorität von v auf p
- `bool PQ.empty()`

```
void DIJKSTRA(const graph& G, node s, const edge_array<int>& cost, node_array<int>& dist,
    node_pq<int> PQ(G);
node v;
forall_nodes(v,G){
    dist[v] = MAXINT
    pred[v] = NULL
}
dist[s] = 0
PQ.insert(s,0);
while(!PQ.empty()){
    node u = PQ.del_min()
```

```

edge e
forall_out_edges(e,u){
    node v = G.target(e)
    int c = dist[u]+cost[e]
    if(c < dist[v]){
        if(dist[v] == MAXINT){
            PQ.insert(v,c)
        }else{
            PQ.decrease(v,c)
        }
        dist[v] = c
        pred[v] = e
    }
}
}

```

Laufzeitanalyse: Operationen auf dem Graphen $\mathcal{O}(n + m)$. PQ-Operationen: 1x Konstruktor, nx insert, nx del.min nx empty, *leqmx* decrease

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

Varianten Datentyp: binärer min-heap, balancierter Baum, Fibonacci-Heap

Varianten Verfahren: All-Pairs-Problem, Single-source-single-target

4 MaxFlow

Transport-Problem: Wie viele Einheiten können von s nach t in einer Zeiteinheit transportiert werden. Jede Kante hat eine Kapazität, die Menge die pro Zeiteinheit über diese Kante laufen kann.

Definition (Network flows): 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), t heißt Senke(target).

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

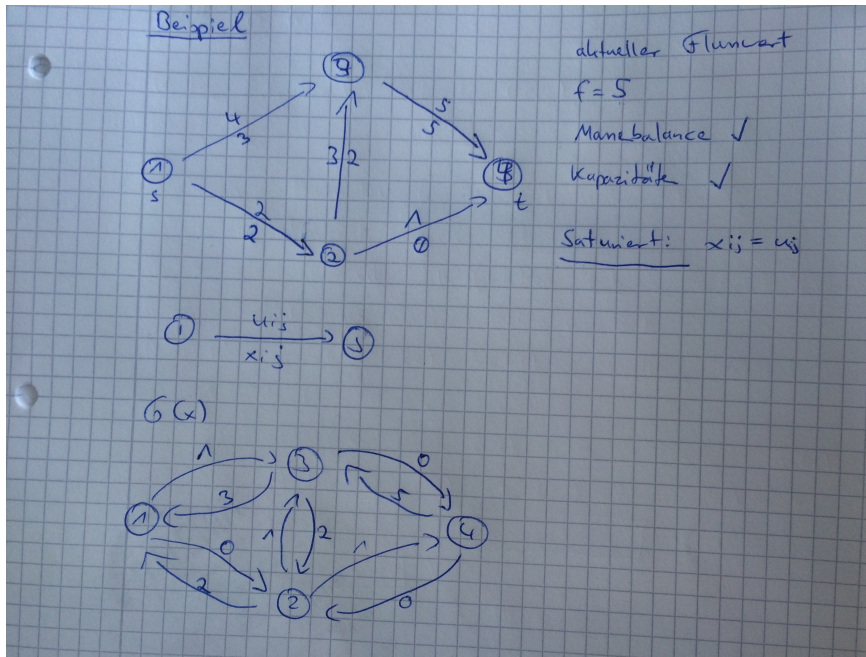
1. $\forall e \in E : 0 \leq x(e) \leq u(e)$ Kapazitätsbedingung
2. Sei $v \in V : \sum_{(v,u) \in E} x(v, u) - \sum_{(w,v) \in E} x(w, v) = \begin{cases} F & : v = s \\ 0 & : v \notin \{s, t\} \\ -F & : v = t \end{cases}$ Mit dem Fluss F . (Massenbalancebedingung)
3. F soll maximal sein.

Bemerkung mit $F=0$ ist das Problem trivial.

Notation: Links der Kante steht die Kapazität, rechts der Fluss.

Knoten $\{1, \dots, n\}$ Kanten (i, j) , Kapazitäten u_{ij} , Flüsse x_{ij}

Definition: Restnetzwerk (residual network): beschreibt mögliche Flussrichtungen. Sei x eine aktuelle Flussfunktion, d.h. x erfüllt die Kapazitätsbedingungen (z.B. Nullfluss $x=0$). Das Restnetzwerk $G(x)$ besteht aus allen Knoten von G und folgenden Kanten: Für jede Kante (i, j) in G existieren in $G(x)$ 2 Kanten (i, j) und (j, i) mit Restkapazitäten $r_{ij} = u_{ij} - x_{ij}$ und $r_{ji} = x_{ij}$. Beschreibt mögliche Flussänderungen der Flussfunktion, ignoriert of alle Kanten mit Restkapazität 0.



Beobachtung: Jeder Pfad in $G(x)$ von s nach t beschreibt eine mögliche Erhöhung des Gesamtflusses f .
 Im Beispiel, ohne 0-Kanten, existieren 3 Pfade, 1-3-2-4 mit den Restkapazitäten 1,2,1. Die mögliche Erhöhung ist das Minimum dieser Pfade.

Wenn in $G(x)$ ohne 0-Kanten kein Pfad von s nach t existiert, ist keine Erhöhung möglich $\Rightarrow f$ maximal.

Für $x = 0$ gilt: $G(x) = G$

Saturation einer Kante (i,j) in $G \Rightarrow x_{ij} \leftarrow u_{ij}$ entfernt (i,j) aus $G(x)$

Auf Null setzen $x_{ij} \leftarrow 0$ entfernt die Gegenkante (i,j) aus $G(x)$.

Bemerkung: Effiziente Implementierungen verwenden später keinen expliziten Restgraphen $G(x)$.

Begriff des Schnitts CUT:

intuitiv: Betrachte einen Schnitt der s und t trennt

Alles was von s nach t fließt, fließt über diesen Schnitt \rightarrow Kapazität des Schnittes ist obere Schranke, das gilt für alle Schnitte und damit auch den minimalen Schnitt.

Formale Definition: (s,t) -Schnitt: Ein (s,t) -Schnitt $[S, \bar{S}]$ ist eine Partitionierung der Knoten V in 2 disjunkte Teilmengen S und $\bar{S} = \{V \setminus S\}$ mit $s \in S$ und $t \in \bar{S}$. Notation: $(i,j) \in (S, \bar{S})$

Fluß von s nach t muss über die Kanten (S, \bar{S}) .

Definition: Kapazität von $[S, \bar{S}]$: $u[S, \bar{S}] = \sum_{(i,j) \in [S, \bar{S}]} u_{ij}$. Analog Restkapazität $r[S, \bar{S}] = \sum_{(i,j) \in [S, \bar{S}]} r_{ij}$. Fluß über den Schnitt $f = \sum_{(i,j) \in [S, \bar{S}]} x_{ij} - \sum_{(i,j) \in [\bar{S}, S]} x_{ij}$, wobei $\sum_{(i,j) \in [S, \bar{S}]} x_{ij}$ maximal $\sum_{(i,j) \in [S, \bar{S}]} u_{ij} = u[S, \bar{S}]$ ist.

Für alle (s,t) -Schnitte gilt $f \leq u[S, \bar{S}]$, insbesondere für den mit minimaler Kapazität. \Rightarrow Der Wert eines maximalen Flusses f ist nicht größer, als die Kapazität eines minimalen Schnittes. ($f_{max} \leq minCut$, schwache Variante von MaxFlowMinCut)

Restnetzwerkberechnen: Explizit teuer, besser implizit.

Algorithmus zur Berechnung des Restnetzwerks:

```
forall_out_edges(e){
    r = cap[e] - flux[e]
    if r==0 continue
    baue Graph
}
forall_in_edges(e){
    r=flow[e]
    if r==0 continue
```

Algorithmus für MaxFlow (Labeling-Algorithmus): 1. $x=0$
 2. while $G(x)$ enthält einen Pfad von s nach t do

3. Sei P ein solcher erhöhender Pfad
4. $S = \min\{r_{ij} | (i, j) \in P\}$
5. Erhöhe den Fluss x entlang von P um r Einheiten.
6. berechne G(x) neu.
7. od

Idee für Implementierung: Sei S die Menge der von s aus erreichbaren Knoten. Wenn t markiert wurde gibt es einen Pfad von s nach t und t liegt in S. Pfade werden in Pred-Array gespeichert.

```

void MF_Labeling(const graph& G, node s, node t, const edge_array<int>& cap, edge_array<int>& pred, const node_array<bool>& labeled) {
    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
                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()
        }
        if (labeled[t]) AUGMENT(G,s,t,pred,cap,flow) //Pfaderhöhung
        else break
    }
}

void AUGMENT(const graph& G, node s, node t, const node_array<edge>& pred, const node_array<bool>& labeled) {
    int delta = MAXINT
    node v = t
    while (v != s){
        int r
        edge e = pred[v]
        if(v == G.source(e)){
            r = flow[e]
            v = G.target(e)
        } else {
            r = cap[e]-flow[e]
            v = G.source(e)
        }
        if(r<delta) delta = r
    }
    v=t
}

```

```

while (v != s) {
    edge e = pred[v]
    if (v == G.source(e)) {
        flow[e] -= delta
        v = G.target(e)
    } else {
        flow[e] += delta
        v = G.source(e)
    }
}
}

```

4.1 Labeling-Algorithmus

Korrektheit

In jedem Schritt (Hauptschleife) gibt es 2 Möglichkeiten:

- a) Algorithmus findet einen erhöhenden Pfad. \rightarrow AUGMENT
- b) Kein erhöhender Pfad (t wird nicht gelabelt) \rightarrow STOP

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

Knoten die gelabelt sind (S) und solche die nicht gelabelt sind (\bar{S}) ist ein (s,t)-Schnitt.

Beobachtung: $\forall (i, j) \in [S, \bar{S}]$ gilt: $v_{ij} = 0$ (deshalb werden diese Kanten vom Algorithmus nicht mehr benutzt, um weitere Knoten zu labeln).

Bei uns bedeutet dies: \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: $F = \sum x_{ij} - \sum x_{kl} (ij) \in (S, \bar{S}), (k, l) \in (\bar{S}, S)$

Hier gilt also: $F = \sum u_{ij} - \sum 0 = u[S, \bar{S}]$

Aus der schwachen Version des Maxflow/MinCut Theorems ($F_{max} \leq U_{min}$) folgt F ist maximal und $u[S, \bar{S}]$ minimal.

Allgemein: Der Algorithmus liefert zusätzlich zur optimalen Lösung des primalen Problems (Maxflow) eine optimale Lösung des dualen Problems (MinCut).

Satz (MaxFlow-MinCut Theorem): Der Wert eines maximalen Flusses ist gleich der Kapazität eines minimalen (s,t)-Schnittes. Beweis durch Korrektheit des Labeling-Algorithmus.

Satz (Augmenting-Path-Theorem): Ein Fluss x ist maximal genau dann, wenn es im Restnetzwerk $G(x)$ keinen erhöhenden Pfad gibt.

Beweis APT: Falls es einen erhöhenden Pfad gibt, ist x nicht maximal. Rückrichtung: \nexists erhöhender Pfad, führe Labeling aus, dann bekommt man einen (s,t)-Schnitt $[S, \bar{S}]$ mit S gelabelt und \bar{S} nicht labeled, mit $F(x) = u[S, \bar{S}]$

Satz (Ganzzahligkeit) Wenn alle Kapazitäten ganzzahlig, dann existiert ein maximaler Fluss x mit $x_{i,j} \in \mathbb{N}$ für all $(i, j) \in E$

Laufzeitanalyse

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

Beobachtung: Höchstens n-1 Kanten verlaufen über den Schnitt (von s nach t). $\Rightarrow F_{max} \leq n * U$. In jeder Iteration von AUGMENT erhöht der Algorithmus den Flusswert F um mindestens 1. Daraus folgt: weniger als F_{max} Iterationen $\leq n * U$.

Eine Iteration (Labeling) kostet Zeit $\mathcal{O}(n + m)$ (Wie DFS/BFS etc.) \Rightarrow Gesamtlaufzeit $\mathcal{O}(n^2 U + nmU)$.

Annahme G ist zusammenhängend $\Rightarrow m \geq n - 1 \rightarrow \mathcal{O}(nmU)$

PREFlow (excess): $e(i) = \sum x_{ji} - \sum x_{ik} \leq 0$

Wenn $e(i) > 0$ dann ist i aktiv ($i \notin \{s, t\}$)

Idee für Algorithmus:

1. Schiebe (push) Fluss von s nach t
2. Während des Ablaufs entsteht dadurch ein Preflow mit ExcessKnoten
3. am Ende soll der Preflow ein echter Fluss sein, d.h. keine aktiven Knoten

Für die Richtung s nach t verwenden wir eine Distanzfunktion (Distanz=Länge von Pfaden von s nach t)

Definition: (Distanz-Label):

Für einen Preflow x definieren wir in $G(x)$ eine Distanzfunktion $d : V \rightarrow \mathbb{N}_0$ mit:

1. $d(t) = 0$
2. $d(i) \leq d(j) + 1$ für alle $(i, j) \in G(x)$

Beobachtung: 1. d ist eine untere Schranke für exakte Distanzwerte. 2. Die Nullfunktion ist ein gültiges Distanz-Label.

Intuition: d = Höhe/Level von Knoten

Definition: admissible/zulässig

Eine Kante (i, j) in $G(x)$ heißt admissible gdw $d(i) = d(j) + 1$

Preflow-Push Algorithmus:

1. Saturiere alle von s ausgehenden Kanten
2. Setze $d(i) = 0$ für $i \neq t$ oder n für $i = s$
3. Betrachte einen aktiven Knoten i und schiebe Fluss über eine admissible Edge.
4. Falls ein aktiver Knoten i keine ausgehende admissible Edge hat, Relabel $d(i) \leftarrow \min\{d(j) \mid j \text{ Nachbar in } G(x)\} + 1$

Der generische Preflow-Push-Algorithmus:

```

1  x=0;e=0;d=0;
2  forall j in V mit (s,j) in E do
3      x_sj = u_sj // Volle Kapazität
4      e(j) += x_sj
5  od
6  d(s) = n
7  while ∃ aktive Knoten do
8      wähle einen aktiven Knoten i // am besten highest label
9      PUSH/RELABEL(i)
10 od

PUSH/RELABEL(knoten i):
1  if ∃ admissible Edge (i,j) in G(x) then
2      wähle eine solche Kante (i,j)
3      delta = min{e(i), r_ij}
4      x_ij += delta
5      e(i) -= delta
6      e(j) += delta
7  else // Relabel
8      d(i) = min{d(j) | (i,j) in G(x)} + 1
9  fi

```

Laufzeit: $\mathcal{O}(\#Push + \#Relabel)$

Lemma: Falls $e(i) > 0$ dann existiert ein Pfad von i nach s im Restnetzwerk $G(x)$.

Lemma2: Für alle aktiven Knoten $i \in V : d(i) < 2n$ folgt aus Lemma 1.

Lemma3: Maximal $2n^2$ Relabels (Folgt aus Lemma2). Für jeden einzelnen Knoten maximal $2n$.

Lemma4: $\leq n * m$ saturierende Pushes

Lemma5: $\mathcal{O}(n^2 * m)$ nicht saturierende Pushes

Der Generische Algorithmus hat $\mathcal{O}(n^2 * m)$