Netzwerkalgorithmen

June 20, 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 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: GRAPH<node_type,edge_type> G
2. Temporäre Daten: besucht[v] ← true

Datentypen in LEDA

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

Anwendung im topologischen Sortieren

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

```
bool TOPSORT(const graph& G, node_array<int>& topnum){
        int count = 0;
        list<node> zerol
        node_array<int> indeg(G);
        node v;
        forall_nodes(v,G){
                indeg[v] = G.indeg(v);
                if(indev[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 $\forall v, w\ inV : v \rightarrow^* w$ (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) meherere 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 dei 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 → Stack eignet sich als Datenstruktur.
Allgemeine Situation für $(v,w) \in T$:
$K' = K_2 \cup K_3 \cup K_4$.

K_1	K_2	K_3	K_4
r_1	r_2	r_3	r_4

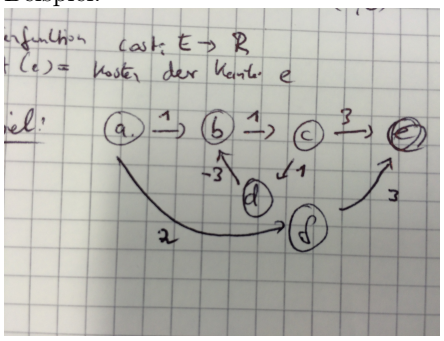
Ergänzungen von DFS für SZK:
1. Aktion: while dfsum[wurzel.top()] > dfsnum[w] do wurzeln.pop() od
2. Falls $(v,w) \in T$: Wurzeln.push(w); Unfertig.push(v)
3. Abschluss eine SZK: SZK von v wird endgültig verlassen, sie ist nun abgeschlossen.

```
Am Ende von dfs(v): if v == Wurzel.top() then
        Wurzeln.pop();
        repeat w = unfertig.pop() until w==v
fi
```

Übung 2: Algo zu SZK.
Darstellung der einzelnen SZKs: node_array< int > szknum(G); k=#kompnum $forall v \in V szknum[v] = i \Leftrightarrow$v in der Komponente i
→ int SZK(const graph& G, node_array< int >& sznum)

3 Kürzeste Wege / Billigste Wege

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



k=3

Kosten eines Pfades P: $cost(P) := \sum_{e \in P} cost(e)$
Billigste-Wege Problem: Finde einen Pfad P mit 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: $dist(a,f) = 2$, $dist(a,e) = -\infty$ weil $a \to b \to (c \to d \to b \to c)^i \to e$
Definition: $dist(v,w) := inf\{cost(P) | P$ist Pfad von v nach w$\}$ d.h. wenn es einen negativen Zyklus gibt $= -\infty$, wenn kein Pfad existiert: ∞.
Wir betrachten hier die Single-Source Variante:
Eingabe: $G = (V,E), s \in V, cost : E \Rightarrow \mathbb{R}$
Ausgabe: $\forall v \in V : dist(s,v)$
Beobachtung: Die dist-Funktion erfüllt die Dreiecksungleichung: $dist(s,w) \leq dist(s,v) + cost(v,w)$

Idee für allgemeinen Algorithmus (Label Correcting):
Überschätze die dist-Werte (DIST=∞). Solange eine Kante (u,v) die Dreiecksungleichung verletzt, korrigiere DIST[v]

```
forall v in V do
        DIST[v] = ∞
od
DIST[s] = 0
while ∃(u,v) ∈ E mit DIST[v] > DIST[u] + cost(u,v)
        DIST[v] = DIST[u]+cost(u,v)
od
```

Eigenschaften

a) Es gilt immer $DIST[v] \geq dist(s,v)$

b) Wenn $DIST[v] < \infty$, dann existiert ein Pfad von s nach v mit den Kosten DIST[v]

c) 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.

d) 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) mit P 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.  forall v ∈ V do
        DIST[v] = ∞
    od
2.  DIST[s] = 0
3.  U = {s}
4.  while U ≠ ∅
5.      wähle und entferne ein u ∈ U
6       forall v∈V mit (u,v)∈E do
7.        c = DIST[u]+cost(u,v)
8.        if c < DIST[v] then
9.              DIST[v]=c
10.             U = U∪ {v}
11.       fi
12.     od
13. od
```

Eigenschaften des Algorithmus während Laufzeit (Lemma):
Falls der Graph G keine negativen Zyklen enthält, dann gilt:
a) Falls $v \notin U$, dann gilt für alle ausgehenden Kanten (v,w): DIST[w]$\leq$ DIST[v]+c(v,w).
b) Sei $v_0, ..., v_k$ 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 $v_i \in U$
c) 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:
a) 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 $s = v_0, ..., v_k = v$ 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.

Perkefte 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[] = 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*(Iterationen über alle Knoten + deren ausgehenden Kanten) = $\mathcal{O}(\sum_{v \in V}(1 + outdeg(v)))$

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

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

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 DIST[u] = dist(s,u)
Beweis: Sei $u \in U$ mit topnum[u] minimal
Annahme: DIST[u] > dist(s,u) $\Rightarrow \exists$ Knoten v auf dem kürzesten Pfad von s nach u mit DIST[v] = dist(s,u) und $v \in U$
$\Rightarrow$ topnum[v] < topnum[u] Widerpsruch.
3. Nicht-Negative Netzwerke: Beliebiger Graph aber Kosten ≥ 0
Dijksra...
Wähle in Zeile 5 einen Knoten $u \in U$ mit DIST[u] minimal. $\Rightarrow$ Priority-Queue.
Behauptung: $u \in U$ mit DIST[u] minimal ist perfekt d.h. DIST[u] = dist(s,u)
Beweis: Annahme: DIST[u] > dist(s,u) $\Rightarrow \exists v \in U$ mit DIST[v] = dist(s,v) auf einem kürzesten Pfad von s nach u.
Dann gilt: dist(s,u) $\geq_{nichtNegativ}$ dist(s,u) $=_{Lemma}$ DIST[v] $\geq_{minmalwahl}$ DIST[u]
$\Rightarrow$ dist(s,u) = 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: foreall_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 Koten u mit DIST[u] minimal.
Datenstruktur: Priority Queue
LEDA: $node_pq < P > PQ$ Knoten-Priority-Queue.
Operationen:

- Konstruktor $node_PQ < P > PQ(GraphG)$ 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
                        }
                }
        }
}
```

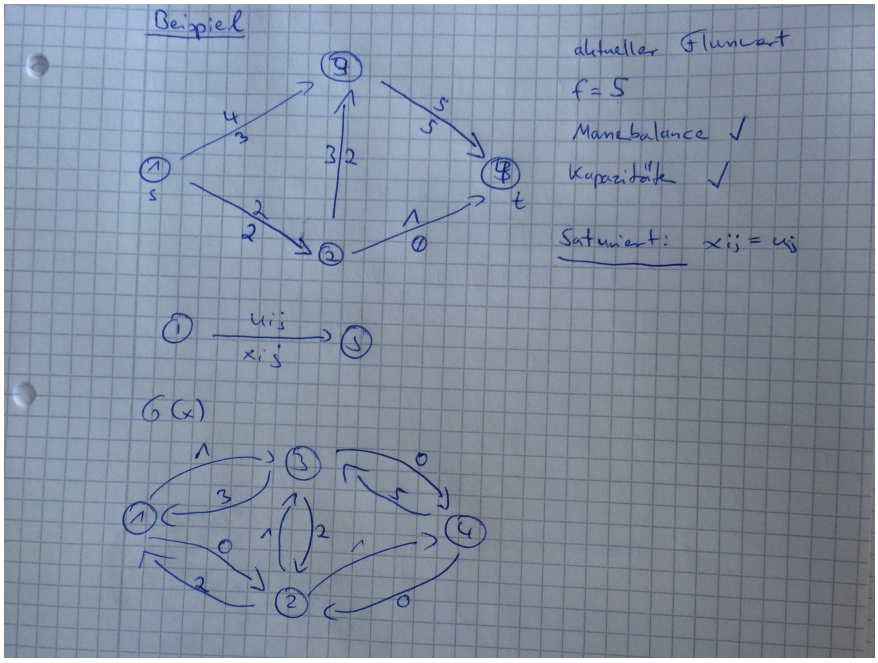
Laufzeitanalyse: Operationen auf dem Graphen $\mathcal{O}(n+m)$. PQ-Operationen: 1x Konstruktor, nx insert, nx del_min nx empty, $leqmx$ decrease
Gesamtlaufzeit: $\mathcal{O}(n*(T_{insert}+T_{delmin}+T_{empty})+m*T_{decrease})$
Varianten Datentyp: binärer min-heap, balancierter Baum, Fibonacci-Heap
Varianten Verfahren: All-Pairs-Problem, Single-source-signle-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 $u: E \to \mathbb{R}_0^+$ 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 $x: E \to \mathbb{R}_0^+$ mit den Eigenschaften:

1. $\forall e \in E: 0 \leq x(e) \leq u(e)$ Kapazitätsbedingung
2. Sei $v \in V: \sum_{(v,u)\in E} x(v,u) - \sum_{(w,v)\in E} x(w,v) = \{F: v=s; 0: v \notin \{s,t\}; -F: v=t\}$ 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,...,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, ingnoriert 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 ⇒ f maximal.
Für x = 0 gilt: G(x)=G
Saturation einer Kante (i,j) in G ⇒ $x_{ij} \leftarrow u_{ij}$ 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 → 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 $[S, \overline{S}]$ ist eine Partitionierung der Knoten V in 2 disjunkte Teilmengen S und $\overline{S} = \{V \setminus S\}$ mit $s \in S$ und $t \in \overline{S}$. NOtation: $(i,j) \in (S, \overline{S})$
Fluß von s nach t muss über die Kanten $(S, \overline{S})$.
Definition: Kapazität von $[S, \overline{S}]$: $u[S, \overline{S}] = \sum_{(i,j)\in[S,\overline{S}]} u_{ij}$. Analog Restkapazität $r[S, \overline{S}] = \sum_{(i,j)\in[S,\overline{S}]}$. Fluss über den Schnitt $f = \sum_{(i,j)\in[S,\overline{S}]} x_{ij} - \sum_{(i,j)\in[\overline{S},S]} x_{ij}$, wobei $\sum_{(i,j)\in[S,\overline{S}]} x_{ij}$ maximal $\sum_{(i,j)\in[S,\overline{S}]} u_{ij} = u[S, \overline{S}]$ ist.
Für alle (s,t)-Schnitte gilt $f \leq u[S, \overline{S}]$, insbesondere für den mit minimaler Kapazität. ⇒ 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) enhält 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<in
        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]) then 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]) then 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) //Pfaderhöung
                else break
        }

}
void AUGMENT( const graph& G, node s, node t, const node_array<edge>& pred, const node_arra
        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

Korektheit

In jedem Schritt (Hauptschleife) gibt es 2 Möglichkeiten:
a) Algorithmus findet einen erhöhenden Pfad. → AUGMENT
b) Kein erhöhender Pfad (t wird nicht gelabeled) → STOP
Zu zeigen: Im Fall b ist der berechnete Fluss maximal.
Knoten die gelabeled sind (S) und solche die nicht gelabeled sind ($\overline{S}$) ist ein (s,t)-Schnitt.
Beobachtung: $\forall (i,j) \in [S,\overline{S}]$ gilt: $v_{ij} = 0$ (deshalb werden diese Kanten vom Algorithmus nicht mehr benutzt, um weitere Knoten zu labeln.
Bei uns bedeutet dies: $\forall$ Kanten $e \in (S,\overline{S})$: $flow[e] == cap[e]$ und $\forall e \in (\overline{S},S) : flow[e] = 0$
Beobachtung über Flüsse und Schnitte: $F = \sum x_{ij} - \sum x_{kl}(ij) \in (S,\overline{S}), (k,l) \in (\overline{S},S)$
Hier gilt also: $F = \sum u_{ij} - \sum 0 = u[S,\overleftarrow{S}]$
Aus der schwachen Version des Maxflow/MinCut Theorems ($F_{max} \leq U_{min}$) folgt F ist maximal und $u[S,\overline{S}]$ 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 maxmial. Rückrichtung: $\not\exists$ erhöhender Pfad, führe Labeling aus, dann bekommt man einen (s,t)-Schnitt $[S,\overline{S}]$ mit S gelabeled und $\overline{S}$ nicht labeled, mit $F(x) = u[S,\overline{S}]$
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 $F_{max} \leq u[S,\overline{S}]$ für ale (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 $\mathcal{O}(n^2U + nmU)$.
Annahme G ist zusammenhängend $\Rightarrow m \geq n-1 \rightarrow \mathcal{O}(nmU)$