Netzwerkalgorithmen

Prof. Dr. Näher

Sommersemester 2016

Inhaltsverzeichnis

1 Definitionen **1**
1.1 Graphen . . . 1
1.1.1 Gerichteter Graph . . . 1
1.2 Mengen . . . 1
1.2.1 Untere Schranke . . . 1
1.2.2 Infimum . . . 1

2 Datentyp für Graphen und Netzwerke **2**
2.1 Definition eines Datentyps . . . 2
2.1.1 Operationen auf G . . . 3
2.1.2 Daten für Knoten und Kanten . . . 4
Algorithmus Topsort . . . 5
Algorithmus Acyclic . . . 6
Algorithmus Tiefensuche (DFS) (rekursiv) . . . 7

3 Starke Zusammenhangskomponenten (SZK) (SCC) **9**
3.1 Definition . . . 9

4 Kürzeste Wege **17**
4.1 Allgemeines Problem . . . 17
4.2 Definitionen . . . 18
4.2.1 Pfad . . . 18
Definition über Knoten . . . 18
Definition über Kanten . . . 18
Länge eines Pfades . . . 18
Einfacher Pfad . . . 18
4.2.2 Kosten eines Pfades . . . 18
4.2.3 Distanz zweier Knoten . . . 19
4.3 Varianten des Problems . . . 19
4.4 Single-Source-Shortest-Path . . . 20
4.4.1 Allgemeiner Algorithmus (Label correcting) . . . 21
4.4.2 Verfeinerter Algorithmus . . . 22

4.5 Perfekte Wahl 25
4.5.1 Allgemeiner Fall 26
Algorithmus von Bellman/Ford 27
4.5.2 Azyklische Graphen 28
Algorithmus Topsort und kürzeste Wege 30
4.5.3 Nicht-negative Netzwerke 31
Algorithmus von Dijkstra 33

5 Maximale Flüsse 35
5.1 Transport-Problem 35
5.1.1 Formale Definition 35
5.1.2 Das Restnetzwerk 36
Definition 37
5.2 Schnitte 41
5.2.1 Definition des (s,t)-Schnittes 41
Kapazität des (s,t)-Schnittes 41
Restkapazität des (s,t)-Schnittes 41
Fluss über den Schnitt 42
Obere Schranke der Kapazität des (s,t)-Schnittes 42
5.3 Algorithmen auf G(x) 42
5.4 Erhöhende-Pfad-Algorithmen 44
5.4.1 Der allgemeine Labeling-Algorithmus 44
5.4.2 Der Labeling-Algorithmus 44
Korrektheit 47
Laufzeitanalyse 49
5.4.3 Algorithmus Capacity-Scaling 51
Korrektheit 53
Laufzeit 53
5.5 Preflow-Push 54
5.5.1 Definition 54
5.5.2 Distanz-Funktion 54
Definition 55
Definition admissible 55
5.5.3 Preflow-Push-Algorithmus 56
Der generische Preflow-Push-Algorithmus 56

A Anhang 59
A.1 Variablenerklärung 59
A.1.1 Kürzeste Wege 59
A.1.2 Maximale Flüsse 60

1 Definitionen

1.1 Graphen

1.1.1 Gerichteter Graph

Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit einer endlichen Menge $V \neq \emptyset$ und einer endlichen Menge $E \subseteq \{(u, v)|u, v \in V, u \neq v\}$. Die Elemente von V heißen Knoten (auf englisch: vertex) von G . Die Elemente von E heißen Kanten (auf englisch: edge) von G. Eine Kante $e \in E$ ist also von der Form $e = (u, v)$ mit $u, v \in V, u \neq v$, wobei u als Anfangsknoten und v als Endknoten von e bezeichnet wird.

1.2 Mengen

1.2.1 Untere Schranke

Sei M eine Teilmenge von $\mathbb{R}$. Dann nennt man eine Zahl u, die kleiner gleich jedem Element von M ist, eine untere Schranke. Es ist also $x \geq u$ für alle $x \in M$.

1.2.2 Infimum

Sei M eine Teilmenge von $\mathbb{R}$. Das Infimum i einer Menge M ist die größte untere Schranke von M. Das Infimum wird charakterisiert über die beiden Eigenschaften:

- Für jedes $y \in M$ ist $y \geq i$.
- Jede Zahl x größer als i ist keine untere Schranke von M: Für alle $x > i$ gibt es mindestens eine Zahl $y \in M$ mit $x > y$.

2 Datentyp für Graphen und Netzwerke

($\rightarrow$ LEDA)

2.1 Definition eines Datentyps

Beispiel 2.1
Ein *stack* ist eine Folge von Elementen vom Typ T.
$Stack < T >$ parametrisierter Typ

- Konstruktion
  $stack < int > S(100)$
  Java: $S = stack < int > (100)$
- Operationen (auf einem Stack S)
  - void S.push(T x)
    Fügt x an der Top-Seite hinzu.
  - T S.pop()
    Entfernt und liefert das Top-Element.
    Precondition: S ist nicht leer.
  - S.empty()
- Iteration $\rightarrow$ Makros
  Liste list<int> L
  $int\ x;\ forall(x, L)\ \{sum+ = x_i\}$
  Makro
  $\#\ define\ forall(x, L)\ for(list_elem\ p = L.first();\ p \neq null;\ p = L.succ(p))$

Bemerkung:
Der allgemeine Graph-Datentyp in LEDA: „*graph*“
Der Datentyp repräsentiert gerichtete Graphen.

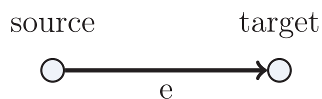
Ein Graph G besteht aus zwei Typen von Objekten:

Knoten vom Typ „*node*“

Kanten vom Typ „*edge*“

Mit jedem Knoten v sind zwei Listen vom Typ $list < edge >$ assoziiert: out_edges und in_edges.

Eine Kante verläuft zwischen genau 2 Knoten.



2.1.1 Operationen auf G

- Update

  $\underline{node}\ G.new_node();$
  Erzeugt einen neuen Knoten in G und gibt ihn (seine Referenz) zurück.
  $\underline{edge}\ G.new_edge(node\ v, node\ w)$
  $\underline{void}\ G.del_edge(edge\ e)$

- Access

  $\underline{list < edge >}\ G.out_edges(node\ v);$
  $\underline{int}\ G.outdeg(node\ v);$
  $\underline{node}\ G.source(edge\ e);$
  $\underline{node}\ G.target(edge\ e);$

- Iteration

$$forall_nodes(v, G)\{\dots\}$$
$$forall_edges(e, G)$$
$$forall_out_edges(e, V)$$
$$forall_in_edges(e, V)$$
$$forall_edges(e, G) \rightarrow node\ v;$$

```
forall_nodes(v, G){
    edge e;
    forall_out_edges(e, v){...}
}
```

2.1.2 Daten für Knoten und Kanten

Es gibt zwei Arten von Daten für Knoten und Kanten.

i) Parametrisierte Graphen: graph<node_type, edge_type>
Beispiel: graph<stadt, autobahn> Verkehrsnetz;
graph<transistor, wire> Schaltkreis;

ii) Temporäre Daten
Beispiel: besucht[v] = true;
dist[u] = 17;

LEDA stellt dafür zwei Datentypen bereit.

- node_array<T> A(G,x) (Initialisierung optional)
Es ist ein Feld über die Knoten des Graphen G.
Der Zugriff erfolgt über A[v].

- edge_array<T> B(G,y)
Es ist ein Feld über die Kanten des Graphen G.
Der Zugriff erfolgt über B[e].

Verwendung:
Dieser Datentyp findet Verwendung für temporäre Daten im Algorithmus und für Eingabedaten und Resultate.

Anwendung im topologischen Sortieren:
Es existiert eine injektive Abbildung $topnum : v \rightarrow \{1 \ldots n\}$
mit $\forall\ (v, w) \in E : topnum[v] < topnum[w]$

Algorithmus Topsort

```
1  bool Topsort(const graph& G, node_array<int>& topnum){
2       int count = 0;
3       list<node> ZERO; // leere Liste von Knoten
4       node_array<int> indeg(G); // aktueller Indegree
5       node v;
6       forall_nodes(v,G){
7            INDEG[v] = indeg(v);
8            if(INDEG[v] == 0)
9                 ZERO.append(v);
10      }
11      while(!ZERO.empty()){
12           u = ZERO.pop();
13           topnum[u] = count+1
14           edge e;
15           forall_out_edges(e,u){
16                v = G.target(e);
17                if(--INDEG[v] == 0)
18                     ZERO.append(v);
19           }
20      }
21      return count == G:number_of_nodes();
22 }
```

Listing 2.1: Topsort

Auf diesem Algorithmus aufbauend lässt sich das Problem lösen, ob ein Graph azyklisch ist.

Gegeben:
Graph G = (V,E)

Idee:
(→ siehe topologische Sortierung)
Entferne jeweils einen Knoten v mit $indeg(v) = \emptyset$ bis der Graph G leer ist.
Falls wir keinen solchen Knoten finden, ist G zyklisch.
Falls G am Ende leer ist, ist G azyklisch.

Algorithmus Acyclic

```
bool Acyclic(graph G){
     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();
}
```

Listing 2.2: Acyclic

Erklärung:
Zuerst werden alle Knoten, die keine eingehenden Kanten besitzen in die Menge $ZERO$ aufgenommen. Danach wird solange noch Knoten in der Menge $ZERO$ enthalten sind, jeweils einer entfernt und für diesen Knoten v überprüft, ob einer der Knoten w, die direkt über eine von v ausgehende Kante mit v verbunden sind, keine eingehenden Kanten mehr hat, wenn diese eine Kante gelöscht würde. Ist dies der Fall, wird w in die Menge $ZERO$ aufgenommen.

Algorithmus Tiefensuche (DFS) (rekursiv)

```
1  void DFS(const graph& G, node_array<int>& dfsnum,
      node_array<int>& compnum){
2       int count1 = 0;
3       int count2 = 0;
4       node_array<bool> visited(G,false);
5       node v;
6       forall_nodes(v,G){
7            if(!visited[v])
8                 dfs(G, v, count1, count2, dfsnum,
                     compnum);
9       }
10 }
```

Listing 2.3: Hauptprogramm

```
1  void dfs(const graph& G, node v; int& count1. int&
      count2, node_array<int>& dfsnum, node_array<int>&
      compnum){
2       dfsnum[v] = ++count1;
3       visited[v] = true;
4       edge e;
5       forall_out_edges(e,v){
6            node w = G.target(e);
7            if(!visited[w])
8                 dfs(G,w,count1, count2, dfsnum, compnum)
9       }
10      compnum[v] = ++count2;
11 }
```

Listing 2.4: Rekursive Funktion

Erklärung:
Zu Beginn des Algorithmus sind alle Knoten als „noch nicht besucht“ markiert.
Das Hauptprogramm ruft für den ersten Knoten v, der noch nicht besucht wurde die rekursive Funktion auf. Diese speichert in der Variable „dfsnum“ zu welchem Zeitpunkt der Knoten v das erste mal besucht wurde und markiert den Knoten v als „besucht“. Dann wird für einen Knoten w, der über eine ausgehende Kante mit v verbunden ist und noch nicht besucht wurde, die rekursive Funktion aufgerufen. Dabei wird aber, anderes als bei der Breitensuche, der nächste von v direkt erreichbare

Knoten erst aufgerufen, wenn alle von w ausgehenden Kanten abgearbeitet wurden. Für seine ausgehenden Kanten gilt aber wieder das gleiche, wie für die ausgehenden Kanten von v.
Sind alle ausgehenden Kanten von v abgearbeitet, wird in der Variablen „compnum“ genau dieser Zeitpunkt gespeichert.

Eine eventuell Bessere Variante wäre die folgende:

```
1  class DFS{
2       aonst graph G,
3       int count1;
4       int count2;
5       node_array<int>&dfsnum;
6       ...
7
8       void dfs(node v){...}
9       void run(){ /* DFS */ }
10      ...
11 }
```

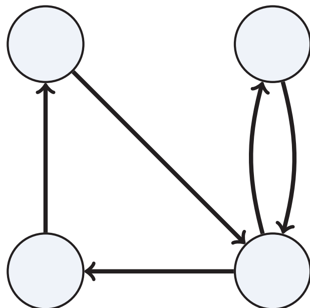
Eine Anwendung von DFS ist die Berechnung von starken Zusammenhangskomponenten.

3 Starke Zusammenhangskomponenten (SZK) (SCC)

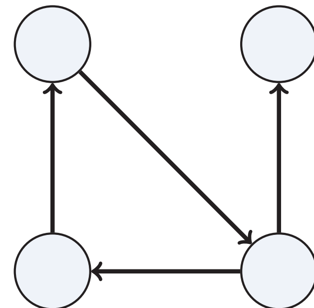
3.1 Definition

a) Ein gerichteter Graph ist stark zusammenhängend $\Leftrightarrow \ \forall\, v, w \in V : v \overset{*}{\to} w$
(es existiert ein Pfad von v nach w)

stark zusammenhängend

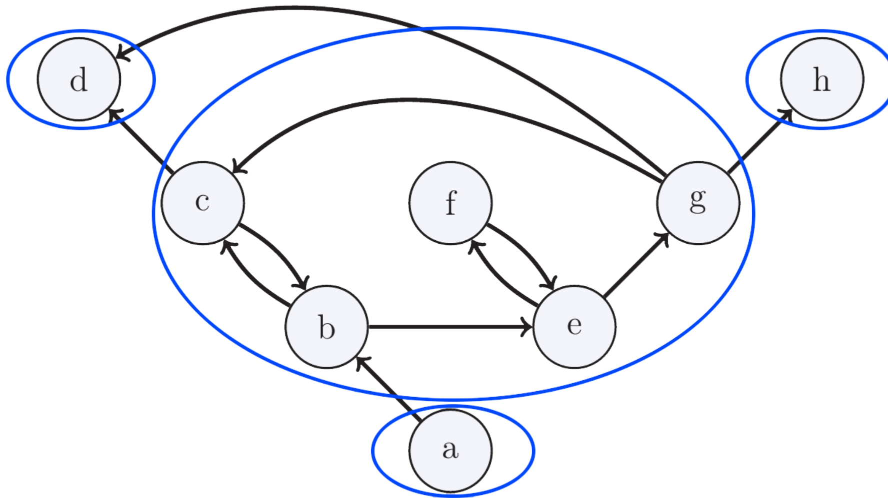


nicht stark zusammenhängend



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

Beispiel 3.1



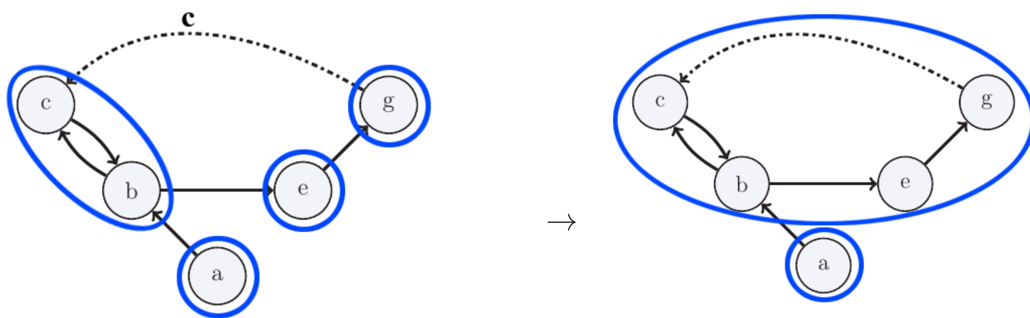
Idee für Algorithmus
Führe DFS aus. Sei $G' = (V', E')$ der Teilgraph aufgespannt von den bereits besuchten Knoten.
Verwalte die SZK von G' (während DFS ausgeführt wird).

Im Beispiel 3.1:
Initialisierung: $V' = \{a\}, E' = \emptyset$
Ablauf:
Sei (v, w) die nächste in dfs betrachtete Kante.

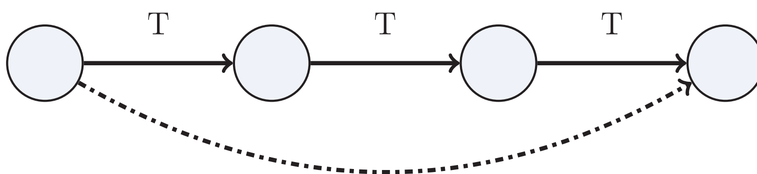
1.Fall $(v, w) \in T$ (Baumkante) (w noch nicht besucht)
$V' = V' \cup \{w\}$
$E' = E' \cup \{(v, w)\}$
$SZK = SZK \cup \{\{w\}\}$
Im Beispiel: $SZK = \{\{a\}, \{b\}\}$

2.Fall $(v, w) \notin T$ dh. w ist bereits besucht ($w \in V'$)
V' unverändert
$E' = E' \cup \{(v, w)\}$
(v, w) kann mehrere SZK zu einer vereinigen
Im Beispiel: Rückwärtskante $(c, b) \in B$

oder Crosskante (Menge C)



Bemerkung:
Vorwärtskanten generieren keine neuen Pfade in G. Deswegen wird SZK nicht geändert. Daher können Vorwärtskanten ignoriert werden.



Definition 3.1

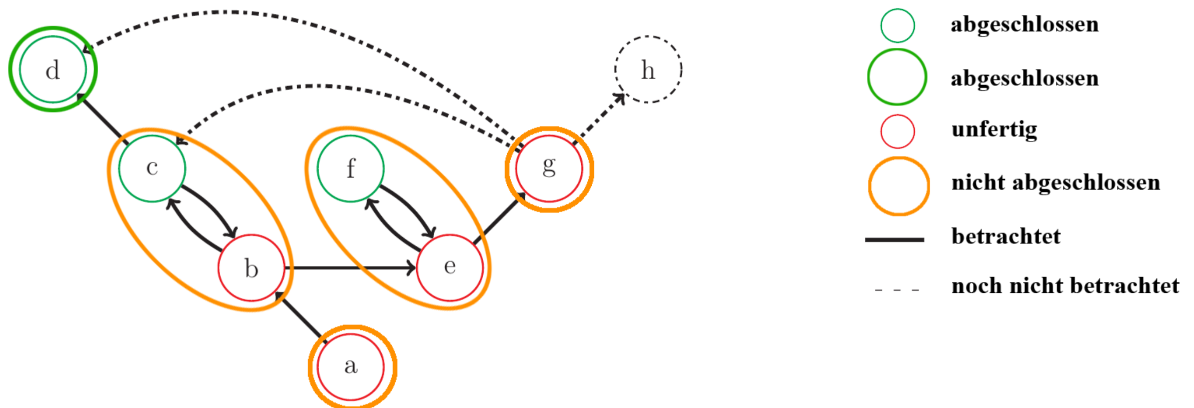
a) Eine SZK K heißt abgeschlossen, falls die Aufrufe von dfs für alle $v \in K$ abgeschlossen sind.

b) Die Wurzel einer SZK K ist der Knoten mit der kleinsten dfsnum in K.

c) Unfertig = Folge aller Knoten für die dfs aufgerufen wurde, aber deren SZK noch nicht abgeschlossen ist. (Sortiert nach dfsnum)

d) Wurzeln = Folge der Wurzeln der nicht abgeschlossenen SZK. (Sortiert nach dfsnum) → Teilfolge von „unfertig“

Im Beispiel 3.1:
Situation, wenn DFS beim Knoten g angelangt ist.



unfertig:	a	b	c	e	f	g
Wurzeln:	a	b		e		g

Der Algorithmus betrachtet danach die Kanten, die aus g heraus gehen.

$(g, d) \in C$
Es passiert nichts, da d in einer abgeschlossenen SZK liegt. ($\Rightarrow$ (g, d) schließt keinen Kreis.)

$(g, c) \in C$
Die Kante vereinigt die drei SZK mit den Wurzeln b,e und g durch streichen von e und g aus der Wurzelfolge.

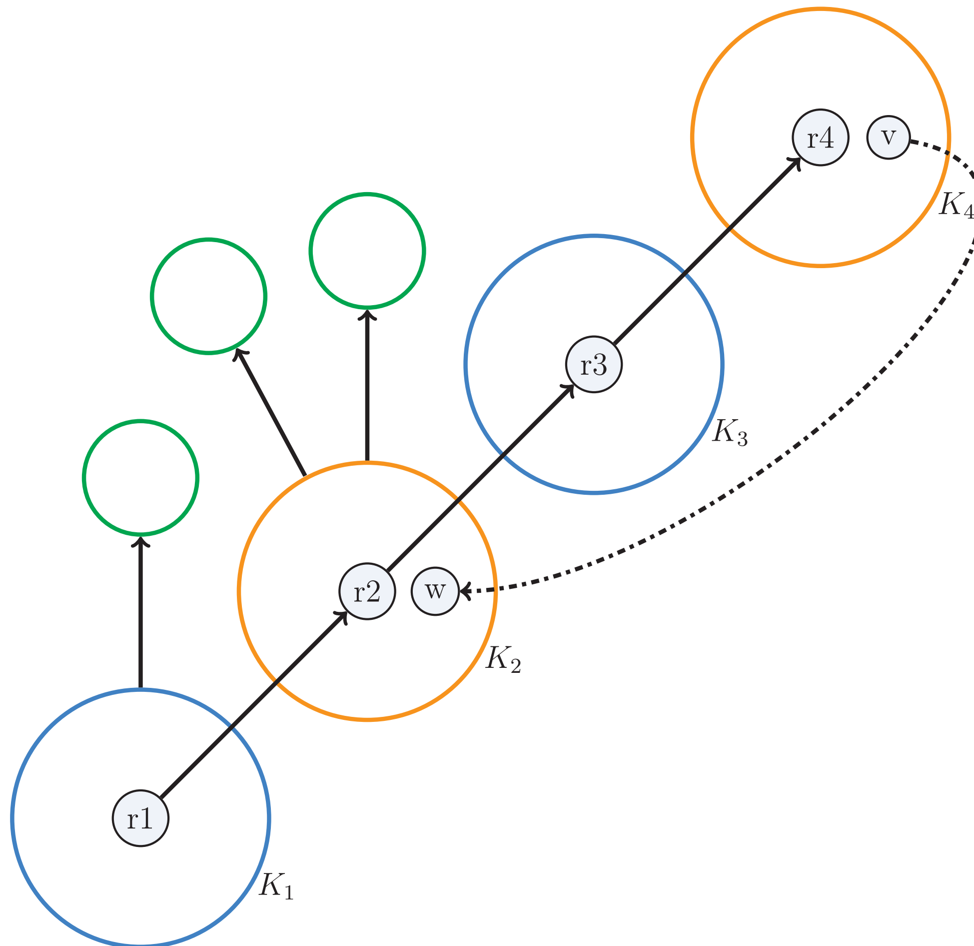
unfertig:	a	b	c	e	f	g
Wurzeln:	a	b				

$(g, h) \in T$
Die Kante fügt den Knoten h an beide Folgen an.

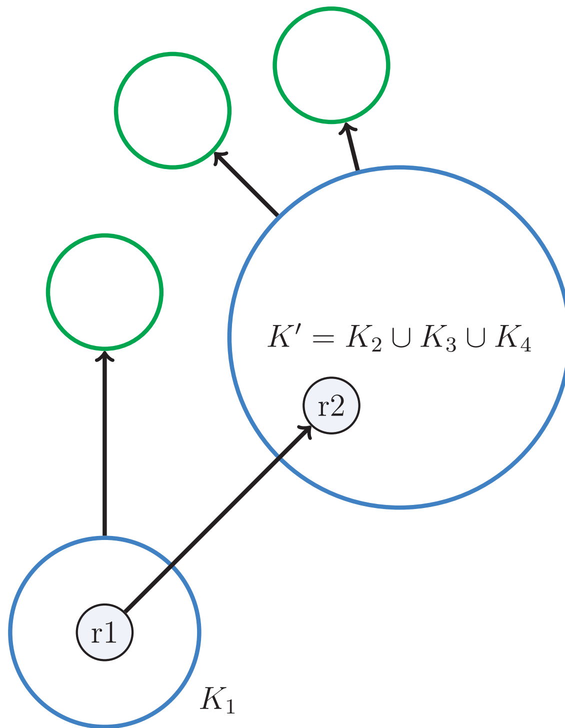
unfertig:	a	b	c	e	f	g	h
Wurzeln:	a	b					h

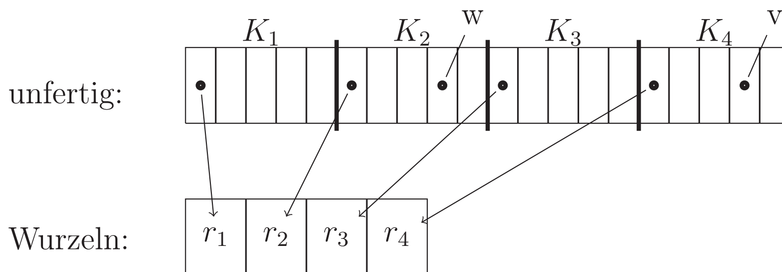
Beobachtung:
Hinzufügen oder Löschen eines Knotens geschieht nur am Ende. Daher nutzt man für die beiden Folgen als Datenstruktur einen Stack.

Allgemeine Situation für $(v, w) \notin T$
w ist unfertig und liegt daher in einer nicht abgeschlossenen SZK.



In diesem Beispiel ist die Kante (v, w) eine Cross- oder Rückwärtskante.
Sie vereinigt dadurch die SZK K_2, K_3 und K_4 zu einer SZK K'.





Aktion:

```
while(dfsnum[wurzeln.top()] > dfsnum[w]){
      wurzeln.pop();
}
```

r_3 und r_4 würden gelöscht.

Falls $(v, w) \in T$
wurzeln.push(v);
unfertig.push(v);

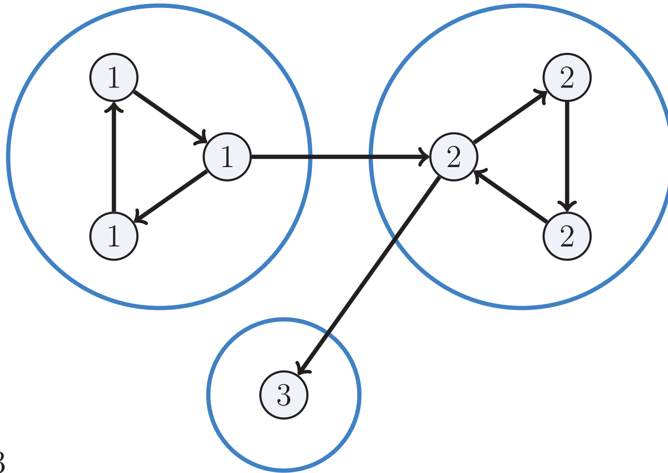
Abschluss einer SZK am Ende von dfs(v)

```
1 if(v == wurzeln.top()){
2       // SZK von v wird endgültig verlassen (dh. abgeschlossen)
3       wurzeln.pop();
4       do {
5             w = unfertig.pop();
6             // Ausgabe
7       }
8       while(w = v)
9 }
```

SZK als C++ Funktion

```
1 int SZK(const graph& Graph, node_array<int>& szknum){
2       // Berechnet das Feld SZKnum
3       // Gibt als Wert die Zahl der SZK zurück
4 }
```

Darstellung der einzelnen SZK's
$node_array < int > SZKnum(G);$
$K = \#$ Komponenten
$\forall\ v \in V\ SZKnum[v] = i \Leftrightarrow\ v$ in der Komponente i. $1 \leq i \leq K$



$K = 3$

4 Kürzeste Wege

Ash Ketchum möchte die kürzeste Strecke von der Universität Trier (Campus 2) zur Pokémon-Arena Porta Nigra bestimmen. Er hat dafür eine Straßenkarte von Trier, auf der die Abstände zwischen allen Paaren benachbarter Kreuzungen eingezeichnet sind.
In diesem Szenario stellen die Kosten einer Kante die Distanz zweier Kreuzungen dar. In anderen Szenarien können die Kosten einer Kante aber auch andere Bedeutungen haben. Sie können z.B. tatsächliche Kosten darstellen, wenn es um Energieverbrauch oder das Herstellen von Produkten geht. Treten in diesen Szenarien negative Kosten auf, können diese Gewinnen entsprechen.

4.1 Allgemeines Problem

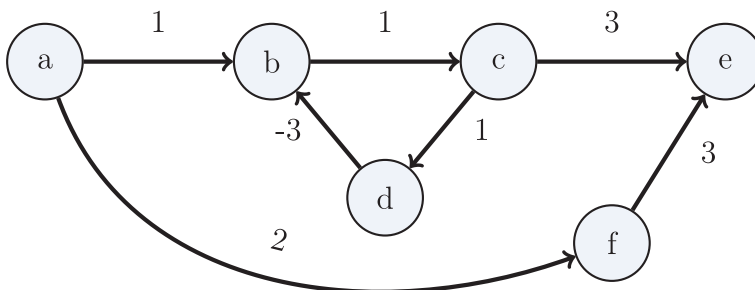
Gegeben:

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

Kostenfunktion: $cost : E \to \mathbb{R}$

$cost(e)$ = Kosten der Kante e
Anstatt $cost((v, w))$ *schreiben wir nur* $cost(v, w)$.

Beispiel 4.1



4.2 Definitionen

4.2.1 Pfad

Ein Pfad kann sowohl über die Knoten, als auch über die Kanten definiert werden, die er enthält.

Definition über Knoten

Ein Pfad von $s \in V$ nach $w \in V$ in einem gerichteten Graphen $G = (V, E)$ ist eine Folge $P = v_0, \ldots, v_k$ mit $v_0 = s$ und $v_k = w$.

Definition über Kanten

Ein Pfad von $s \in V$ nach $w \in V$ in einem gerichteten Graphen $G = (V, E)$ ist eine Folge $P = (e_1, e_2, \ldots, e_n)$ von Kanten $e_i \in E$.

Länge eines Pfades

Die Länge eines Pfades wird oft über die Anzahl der enthaltenen Kanten definiert. Hier allerdings **nicht**!
Trotzdem sprechen wir von kürzesten Pfaden, da das Problem im Englischen „shortest path“ genannt wird und es oft mit „kürzestem Weg“ übersetzt wird. Da oft von billigsten Pfaden gesprochen wird, wenn es um die Kosten eines Pfades geht, werden in diesem Skript die Bezeichnungen „kürzeste Wege“ und „billigste Pfade“ (oder Mischformen daraus) synonym verwendet.

Einfacher Pfad

Ein einfacher Pfad enthält keinen Knoten mehrfach.
Erfüllt ein Pfad diese Eigenschaft nicht nennt man ihn „nicht einfachen Pfad“.

4.2.2 Kosten eines Pfades

Die Kosten eines Pfades ist die Summe der Kosten der Kanten entlang des Pfades.

$$\boldsymbol{cost(P) := \sum_{e \in P} cost(e)}$$

Achtung: Eine Kante kann mehrfach vorkommen.

Im Beispiel 4.1:

$$cost(a \to f \to e) = 5$$
$$cost(a \to b \to c \to e) = 5$$
$$cost(a \to b \to c \to d \to b \to c \to e) = 4$$

4.2.3 Distanz zweier Knoten

Die Distanz zweier Knoten entspricht dem kürzesten Weg zwischen diesen beiden Knoten.
Da es durch negative Zyklen unendlich viele Wege geben kann, wird hier nicht das Minimum, sondern das Infimum (s. 1.2.2) genutzt.

$\boldsymbol{dist(v, w) := inf\{cost(P) \mid P}$ **ist Pfad von v nach w}**

dh. $dist(v, w) = -\infty$, falls negativer Zyklus auf P existiert.
$dist(v, w) = +\infty$, falls kein Pfad von v nach w existiert.

Im Beispiel 4.1:

$$dist(a, f) = 2$$
$$dist(e, a) = +\infty$$
$$dist(a, e) = -\infty \qquad \text{Pfad: } a \to b \to (c \to d \to b \to c)^i \to e$$

Kosten: $4 - i$ dh. beliebig klein.

4.3 Varianten des Problems

Es existieren verschiedene Varianten des Kürzesten-Wege Problems. Dabei wird unterschieden, wie viele source und target Knoten es gibt.

i) **Single-Source-Single-Target**
$dist(s, t) \quad s, t \in V$
Zwischen zwei gegebenen, festen Knoten wird die Distanz gesucht.

ii) **Single-Source-Shortest-Path**
$dist(s, v) \quad \forall\, v \in V$
Von einem gegebenen Knoten s werden die kürzesten Distanzen zu allen anderen Knoten gesucht.

iii) **All-Pairs-Problem**
$dist(v, w) \quad (v, w) \in VxV$
Gesucht ist die Distanz zwischen allen Knotenpaaren. (→ Entfernungstabelle)

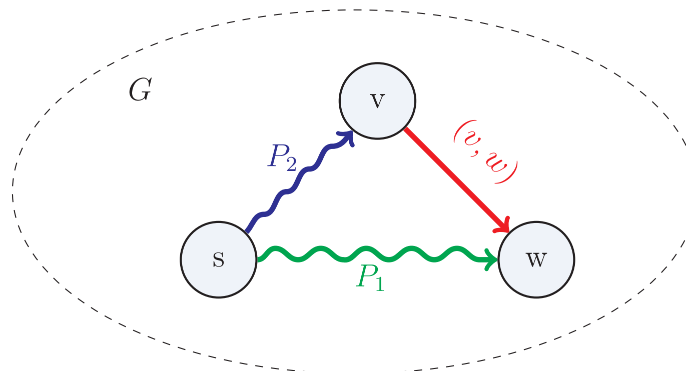
4.4 Single-Source-Shortest-Path

Ab hier wird das Single-Source-Shortest-Path Problem betrachtet.

Eingabe: $G = (V, E), \quad s \in V, \quad cost : E \to \mathbb{R}$
Ausgabe: $\forall\, v \in V \quad dist(s, v)$ (und entsprechende Pfade)

Beobachtung:
Die *dist*-Funktion erfüllt die Δ-Ungleichung.
Für jede Kante $(v, w) \in E$ gilt:

$$dist(s, w) \leq dist(s, v) + cost(v, w)$$



P_1 ist ein kürzester Pfad von s nach w.
P_2 ist ein kürzester Pfad von s nach v.

Herleitung der Δ-Ungleichung:
$dist(s, w) = cost(P_1)$
$dist(s, v) = cost(P_2)$

$dist(s, w) \leq cost(P_2 + (v, w))$ da P_1 kürzester Pfad von s nach w
$\Leftrightarrow dist(s, w) \leq cost(P_2) + cost(v, w)$
$\Leftrightarrow dist(s, w) \leq dist(s, v) + cost(v, w$

4.4.1 Allgemeiner Algorithmus (Label correcting)

Aus diesen Beobachtungen kann leicht ein Algorithmus abgeleitet werden.

Ideen:

- Überschätze die $dist$-Werte ($DIST = \infty$)
- Solange eine Kante (u, v) die Δ-Ungleichung verletzt: Korrigiere $DIST[v]$ dh. $DIST[v] \leftarrow DIST[u] + cost(u, v)$

Algorithmus 1 : Allgemeiner Algorithmus

```
1 foreach v ∈ V do
2 |   DIST[v] ← ∞
3 end
4 DIST[s] ← 0
5 while ∃ Kante (u, v) ∈ E mit DIST[v] > DIST[u] + cost((u, v)) do
6 |   DIST[v] ← DIST[u] + cost((u, v))
7 end
```

Lemma 1

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

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

c) *Die kürzesten Pfade bilden einen Baum* T *mit Wurzel* s*.*

d) *Für jede Kante* (v, w) *auf einem kürzesten Pfad gilt:*
$dist(s, w) = dist(s, v) + cost((v, w))$ *(Δ-Ungleichung mit Gleichheit)*

Erklärungen zu Lemma 1:

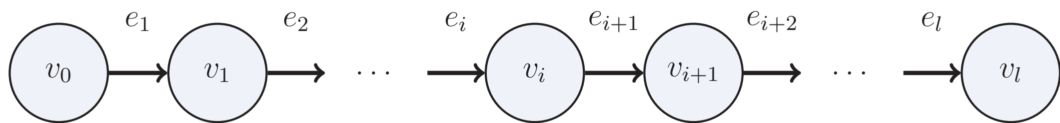
a) $dist(s, v)$ sind die Kosten des kürzesten Pfades von s nach v. $DIST[v]$ ist zu Beginn $+\infty$ und wird im Laufe des Algorithmus, falls mindestens ein Pfad von s nach v existiert, auf die Kosten dieses Pfades verringert. Somit kann

der Wert von $DIST[v]$ auch nie kleiner als die Kosten des kürzesten Pfades ($dist(s, v)$) werden. Ab dem Zeitpunkt, an dem der Algorithmus den kürzesten Pfad findet, gilt: $DIST[v] = dist(s, v)$

b) Sobald der Algorithmus einen Pfad von s nach v findet, berechnet er die Kosten dieses Pfades und setzt dieses Ergebnis als neuen $DIST$-Wert von v. Da der Pfad und die Kosten der einzelnen Kanten endlich sind, ist das Ergebnis und somit $DIST[v]$ echt kleiner als $+\infty$.

c) In jedem Knoten mit $DIST[v] < \infty$ mündet genau ein kürzester Pfad. Bei Korrektur des $DIST$-Wertes wird die eingehende Kante von T definiert.

d) Die Δ-Ungleichung mit Gleichheit ergibt sich aus der Definition von $dist(s, w)$.



$$dist(s, v_i) = \sum_{k=1}^{i} e_k$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i} e_k + cost(v_i, v_{i+1})$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i} e_k + e_{i+1}$$

$$dist(s, v_{i+1}) = \sum_{k=1}^{i+1} e_k$$

(v_i, v_{i+1}) verbindet den Knoten v_{i+1} mit dem auf dem kürzesten Weg davor liegenden Knoten.

4.4.2 Verfeinerter Algorithmus

Der allgemeine Algorithmus lässt viel Freiheit bei der Wahl des Knotens. Dadurch kann der Knoten sehr ungeschickt gewählt werden. Liegt der Knoten sehr weit hinten im Graph, müssen die $DIST$-Werte der hinteren Knoten immer wieder angepasst werden, wenn bei einem auf dem Pfad davor liegenden Knoten der $DIST$-Wert verringert wurde. Daher beginnt der folgende Algorithmus beim Startknoten, sodass

die $DIST$-Werte wie eine Welle durch den Graphen vom Startknoten aus verändert werden.

Damit der Algorithmus nicht mehr jedes mal alle Kanten überprüfen muss, wird eine Kandidatenmenge gebildet. In dieser Kandidatenmenge U werden alle Knoten, aus denen Kanten ausgehen, die die Δ-Ungleichung verletzen können, gespeichert. Dadurch muss der Algorithmus immer nur die Kandidatenmenge abarbeiten und arbeitet somit effizienter.
Am Anfang besteht U nur aus dem Startknoten s. $\quad U = \{s\}$
Immer, wenn $DIST[v]$ vermindert wird, wird v in die Menge U aufgenommen, da die Δ-Ungleichung verletzt worden sein könnte.

Algorithmus 2 : Verfeinerter Algorithmus

1 **foreach** $v \in V$ **do**
2 $\quad DIST[v] \leftarrow \infty$
3 **end**
4 $DIST[s] \leftarrow 0$
5 $U = \{s\}$
6 **while** $U \neq \emptyset$ **do**
7 $\quad$ wähle und entferne ein $u \in U$
8 $\quad$ **foreach** $v \in V$ *mit* $(u, v) \in E$ **do**
9 $\quad\quad c = DIST[u] + cost((u, v))$
10 $\quad\quad$ **if** $c < DIST[v]$ **then**
11 $\quad\quad\quad DIST[v] = c$
12 $\quad\quad\quad U = U \cup \{v\}$
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**

Lemma 2

Falls G keine negativen Zyklen enthält, dann gilt folgendes:

a) Falls $v \notin U$, dann gilt für alle ausgehenden Kanten (v, w):
$DIST[w] \leq DIST[v] + c((v, w))$ (dh. Δ-Ungleichung erfüllt)

b) Sei $v_o, \ldots, v_k$ ein kürzester Pfad von s nach v (dh. $v_0 = S$, $v_k = v$).
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 $v_i \in U$ z.B. $s = 0$

c) Es existiert immer ein $u \in U$ mit $DIST[u] = dist(s, u)$

d) Wenn in Zeile 7 des Algorithmus 2 stets ein $u \in U$ mit $DIST[u] = dist(s, u)$ gewählt wird ("perfekte Wahl"), dann wird die while-Schleife für jeden Knoten höchstes einmal ausgeführt.

Beweis

a) Induktion über die Schleifendurchläufe (while):

$\underline{i = 0}$: Vor dem ersten Lauf $DIST[s] = 0$, $DIST[v] = \infty$ für $v \neq s$ und $U = \{s\}$

$\underline{i \to i+1}$: Betrachte ein beliebiges $v \notin U$ nach der $(i+1)$ten Ausführung.

Fall 1: $v \notin U$ vor $(i+1)$ten Ausführung. Nach I.A. galt:
$DIST[w] \leq DIST[v] + c((v, w))$ für alle ausgehende Kanten (v, w).
$DIST[v]$ wurde im $(i+1)$ten Lauf nicht verändert(da $v \notin U$ danach) und die $DIST$-Werte aller Nachbarn w von v wurden eventuell vermindert
$\Rightarrow DIST[w] \leq DIST[v] + c((v, w))$ für alle ausgehenden Kanten (nach der $(i+1)$ten Ausführung)

Fall 2: $v \in U$ vor $(i+1)$Ausführung
$\Rightarrow v$ wurde in Zeile 7 ausgewählt (dh. $u = v$)
$\Rightarrow$ Die innere Schleife stellt die Δ-Ungleichung für alle ausgehenden Kanten her.

b) Sei $s = v_0, \ldots, v_k = v$ ein kürzester Pfad von s nach v mit $DIST[v_k] > dist(s, v_k)$.
Sei i maximal mit $DIST[v_i] = dist(s, v_i) \Rightarrow 0 \leq i < k$
Z.z. $v_i \in U$
Widerspruchsbeweis:
Annahme: $v_i \notin U$

Teil a) $\Rightarrow DIST[v_{i+1}] \leq DIST[v_i] + c((v_i, v_{i+1}))$
Außerdem gilt: $dist(s, v_{i+1}) = dist(s, v_i) + c((v_i, v_{i+1}))$ (auf kürzesten Pfaden ist die Δ-Ungleichung mit Gleichheit erfüllt)
Dann folgt:

$$\begin{aligned} dist(s, v_{i+1}) &= dist(s, v_i) + c((v_i, v_{i+1})) \\ &= DIST[v_i] + c((v_i, v_{i+1})) \\ &\geq DIST[v_{i+1}] (\text{da } v_i \notin U) \end{aligned}$$

Zusammen: $dist(s, v_{i+1}) \geq DIST[v_{i+1}]$
$\Rightarrow dist(s, v_{i+1}) = DIST[v_{i+1}]$ ↯ Widerspruch zur maximalen Wahl von i da $DIST$-Werte nie zu klein werden.

c) Sei $v \in U$ beliebig.

Fall 1: $DIST[v] = dist(s, v) \Rightarrow$ fertig.

Fall 2: $DIST[v] > dist(s, v)$
Betrachte einen kürzesten Pfad von s nach v. Dann existiert auf diesem Pfad (nach Teil b)) ein $v_i \in U$ mit $DIST[v_i] = dist(s, v_i)$.

d) Falls in Zeile 7 des Algorithmus 2 immer eine perfekte Wahl getroffen wird, dann kann u kein zweites Mal in U eingefügt werden.

Perfekte Wahl
$\rightarrow$ Gesamtaufwand: $\mathcal{O}\Big(\underbrace{\sum_{v \in V}(1 + outdeg(v))}_{n+m} + \underbrace{\text{Verwaltung der Menge } \mathcal{U}}_{?}\Big)$

$n = \#$Knoten
$m = \#$Kanten

□

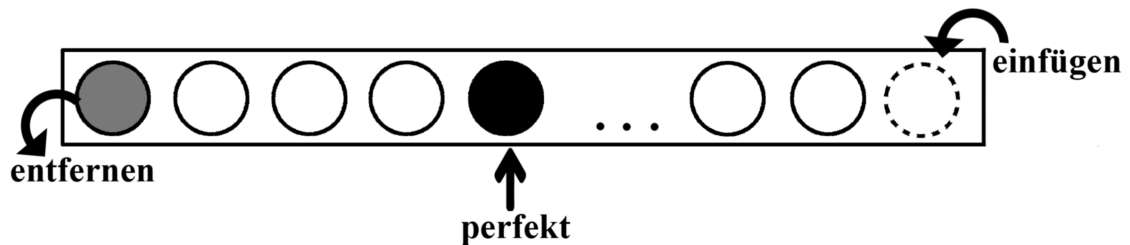
4.5 Perfekte Wahl

Frage: Wie findet man effizient ein perfektes u, um es in Zeile 7 des Algorithmus 2 zu wählen?

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.
$\Rightarrow$ Zwischen zwei Entnahmen des selben Knoten u wurde mindestens ein perfekter Knoten entfernt. Dieser kann nicht wieder in die Schlage eingefügt werden.
$\Rightarrow$ 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.
$\rightarrow$ Algorithmus von Bellman/Ford

Algorithmus von Bellman/Ford

```
bool BELLMANFORD(const graph& G, node s, const
   edge_array<int>& cost, node_array<int>& DIST,
   node_array<edge>& PRED){
      queue<node> Q;
      node_array<bool> inQ(G,false);
      node_array<int> count(G,0);
      node v;
      forall_nodes(v,G){
            DIST[v] = MAXINT;
            PRED[v] = NULL;
      }
      DIST[s] = 0;
      Q.append(s);
      inQ[s] = true;
      while(!Q.empty()){  // Solange Q nicht leer ist
            node u = Q.pop()
            inQ[u] = false;
            if(++count[u] > G.numbers_of_nodes())
                  return false;
            edge e;
            forall_out_edges(e,u){
                  node v = G.target(e);
                  int d = DIST[u] + cost(e);
                  if(d < DIST[v]){
                        DIST[v] = d;
                        PRED[v] = e;
                        if(!inQ[v]){
                              Q.append(v);
                              inQ[v] = true;
                        }
                  }
            }
      }
      return true;
}
```

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:

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

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

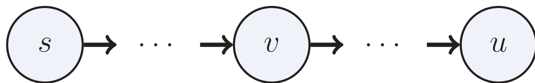
In azyklischen Graphen treten keine Zyklen auf. Daher ist es auch, trotzt beliebiger Kostenfunktion und dadurch möglicher negativer Kosten, nicht möglich, dass negative Zyklen auftreten. Dadurch existiert eine topologische Sortierung der Knoten.

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: $DIST[u] > dist(s, u)$
$\xRightarrow{\text{Lemma 2 b)}}$ $\exists$ Knoten v auf dem kürzesten Pfad von s nach u mit:

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

2) $v \in U$



$\Rightarrow topnum[v] < topnum[u]$ ↯
Widerspruch zur kleinsten Wahl von u (kleinste $topnum$ in U)
$\Rightarrow DIST[u] = dist(s, u)$ □

Mögliche Implementierungen:

1) Topsort → Liste aller Knoten in U mit aufsteigenden *topnum*‘s
   Durchlaufe diese Knoten

2) Kombinierter Algorithmus aus Topsort und kürzeste Wege

Algorithmus Topsort und kürzeste Wege

```
void Topsort(const graph& G, node s, const edge_array<
   int> cost, node_array<int>& Dist){
     node_array<int> INDEG(G,0);
     forall_nodes(v,G){
          INDEG[v] = indeg(v);
          if(INDEG[v] == 0)
               ZERO.append(v);
          DIST[v] = MAXINT;
     }
     DIST[s] = 0;
     while(!ZERO.empty()){
          u = ZERO.pop();
          edge e;
          forall_out_edges(e,u){
               v = G.target(e);
               if(--INDEG[v] == 0)
                    ZERO.append(v);
               int d = DIST[u] + cost(u,v);
               if(d < DIST[v])
                    DIST[v] = d;
          }
     }
}
```

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 $PQ.delmin()$.
$\rightarrow$ 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:

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

$\Rightarrow dist(s, u) = DIST[u]$ da $DIST$-Werte nie zu klein werden. ↯ 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: $I = node$
$P = int\ DIST$-Werte

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

Operationen:

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

Algorithmus von Dijkstra

```
void Dijkstra(const graph& G, node s, const edge_array<
   int>& cost, node_array<int>& DIST, node_array<edge>&
    PRED){
     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(); // perfekte Wahl
          egde e;
          forall_out_edges(e,u){
               node v = G.target(e);
               int d = DIST[u] + cost[e];
               if(d < DIST[v]){
                    if(DIST[v] == MAXINT)
                         PQ.insert(v,d);
                    else // v ist in U
                         PQ.decrease_p(v,d)
                    DIST[v] = d;
                    PRED[v] = e;
               }
          }
     }
}
```

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: $\mathcal{O}(n+m)$

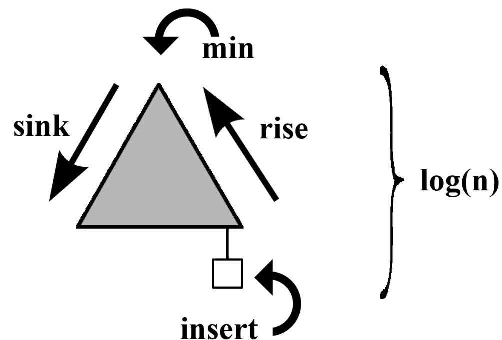
ii) Priority Queue:
$1 \cdot$ Konstruktor
$n \cdot insert()$
$n \cdot delmin()$
$n \cdot empty()$
$m \cdot decrease()$

$\Rightarrow$ Gesamtlaufzeit: $\mathcal{O}\big(n \cdot (T_{insert}(n) + T_{delmin}(n) + T_{empty}(n)) + m \cdot T_{decrease}(n)\big)$
$T_{op}(n) =$ 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 $u : E \to \mathbb{R}_0^+$ und
- zwei Knoten s, $t \in V$ mit $s \neq t$.

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

Gesucht ist eine Flussfunktion $x : E \to \mathbb{R}_0^+$ 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!)

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

$F \geq 0$

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.

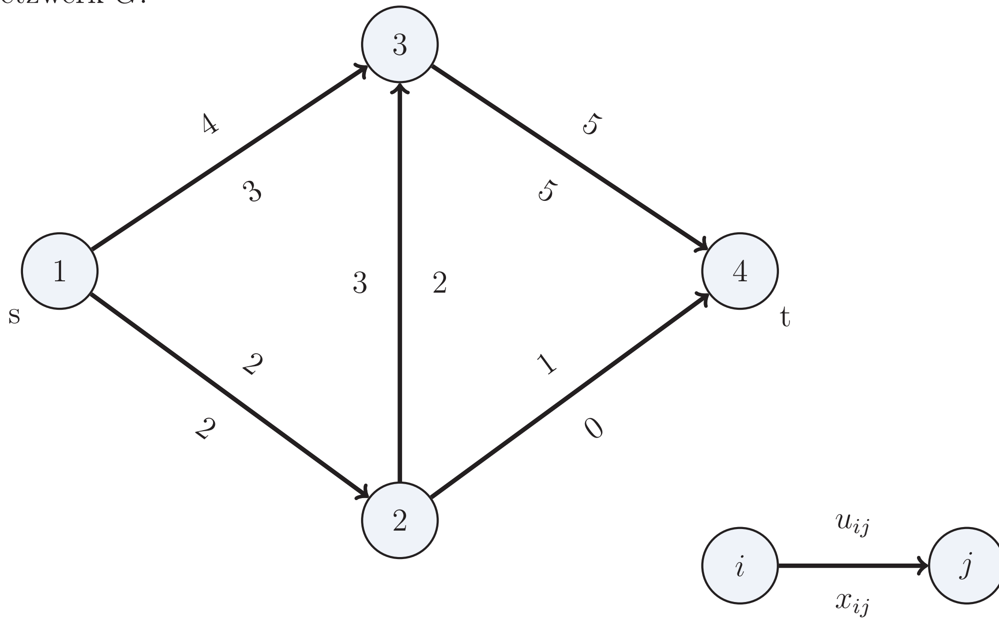
Definition

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 $r_{ji} = x_{ij}$.

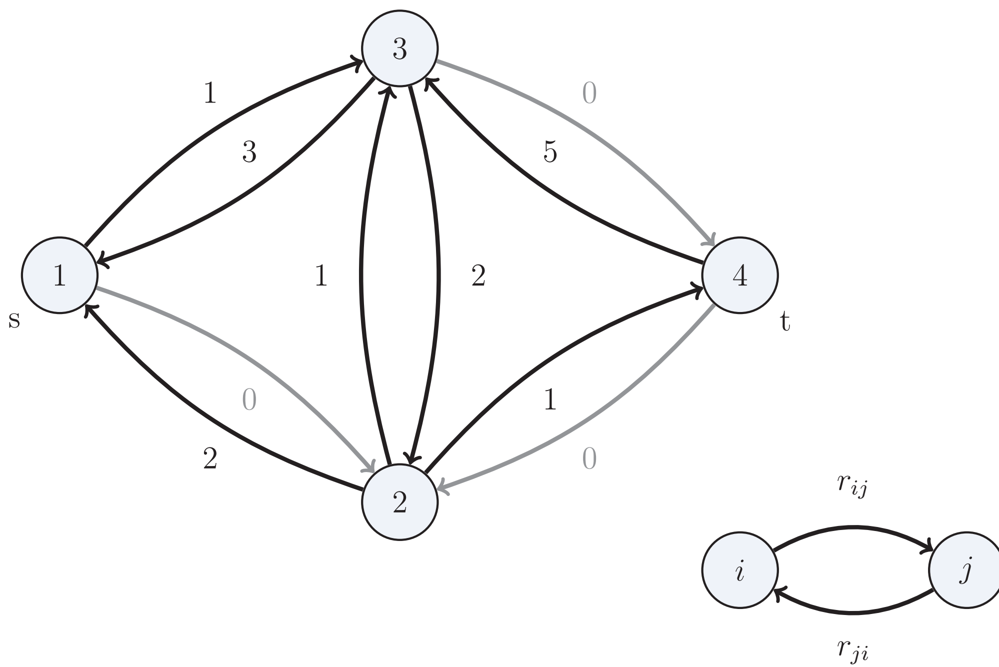
Die Restkapazitäten beschreiben mögliche Änderungen an der Flussfunktion. Eine Erhöhung geschieht in Richtung der Kante (i, j) (max $r_{ij} = u_{ij} - x_{ij}$) und eine Verminderung in die Gegenrichtung (max $r_{ji} = x_{ij}$).

Beispiel 5.1

Netzwerk G:



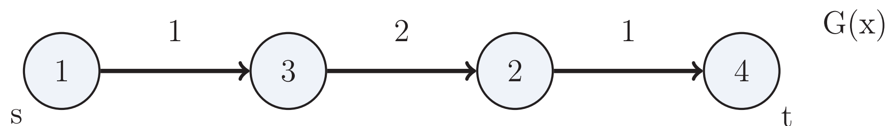
Aktueller Flusswert: $F = 5$
Restnetzwerk $G(x)$:



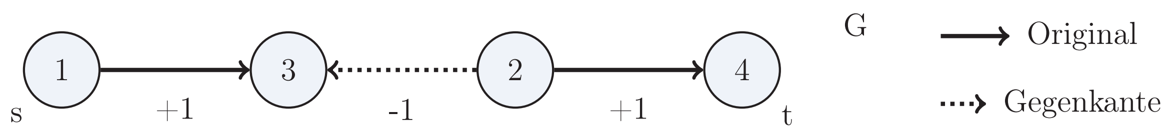
Da über Kanten mit einer Restkapazität von Null, nichts mehr fließen kann, werden sie weggelassen. Dadurch ist die folgende Beobachtung möglich.

Beobachtung: (ohne 0-Kanten)
Wenn alle Null-Kanten im Graphen weggelassen werden, entspricht die Frage ob ein Pfad existiert der Frage ob ein erhöhender Pfad existiert.
Jeder Pfad in $G(x)$ von s nach t beschreibt eine mögliche Erhöhung des Gesamtflusses F. ($\rightarrow$ Pfaderhöhung)

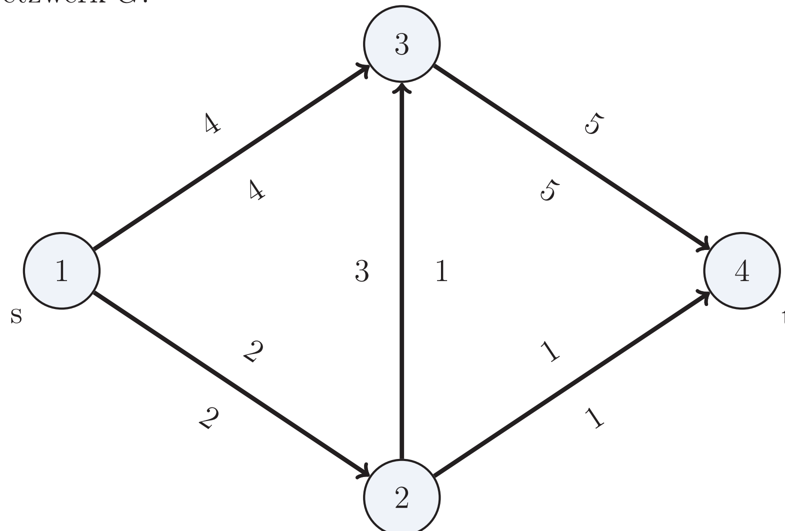
Im Beispiel 5.1 existiert nur ein solcher Pfad:



Eine Erhöhung um $\delta = min\{v_{ij} | (i,j) \in P\} = 1$ ist möglich.

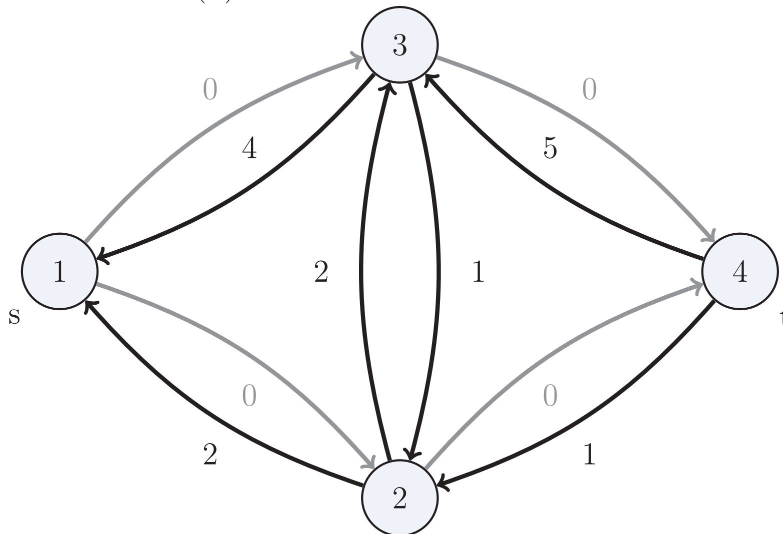


Netzwerk G:



Flusswert $F = 6$

Restnetzwerk $G(x)$:



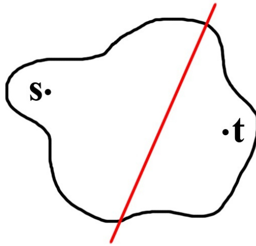
Hier existiert in $G(x)$ kein Pfad mehr von s nach t. Daher ist keine weitere Erhöhung mehr möglich.

Anmerkungen:

- Für den Nullfluss ($x = 0$) gilt $G(x) = G$.
- Wird eine Kante (i, j) in G saturiert (dh. $x_{ij} = u_{ij}$), so wird die Kante (i, j) aus $G(x)$ entfernt.
- Setzt man $x_{ij} = 0$, so wird die Gegenkante (j, i) aus $G(x)$ entfernt.

$\Rightarrow$ Änderungen an der Flussfunktion führen zu Änderungen in $G(x)$.

5.2 Schnitte

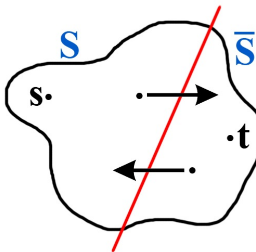


Betrachte einen Schnitt der s und t trennt. Alles was von s nach t fließt, fließt auch über den Schnitt.
⇒ Kapazität des Schnittes ist obere Schranke für den maximalen Fluss.
Das gilt für alle Schnitte und damit auch für den minimalen Schnitt ($MinCut$).

5.2.1 Definition des (s,t)-Schnittes

Ein (s,t)-Schnitt [S,$\bar{S}$] ist eine Partitionierung der Knoten V in zwei disjunkte Teilmengen S und $\bar{S}$ ($= V \setminus S$) mit $s \in S$ und $t \in \bar{S}$.

Notation:



$(i, j) \in (S, \bar{S})$ oder $(i, j) \in (\bar{S}, S)$.

Der Fluss von s nach t muss über die Kanten $\underbrace{(S, \bar{S})}_{Vorwärtskanten}$ $(= E \cap S \times \bar{S})$

$(\bar{S}, S)$ sind die Rückwärtskanten.

Kapazität des (s,t)-Schnittes

Die Kapazität des (s, t)-Schnittes ist die Summe der Kapazitäten der Kanten, deren source-Knoten in S und deren target-Knoten in $\bar{S}$ liegen.

$$u[S, \bar{S}] = \sum_{(i,j) \in (S, \bar{S})} u_{ij}$$

Restkapazität des (s,t)-Schnittes

Die Restkapazität des (s, t)-Schnittes ist die Summe der Restkapazitäten der Kanten, deren source-Knoten in S und deren target-Knoten in $\bar{S}$ liegen.

$$r[S, \bar{S}] = \sum_{(i,j) \in (S,\bar{S})} r_{ij}$$

Fluss über den Schnitt

Der Wert des Flusses über den Schnitt ist die Differenz zwischen der Summe der Flusswerte der Kanten, die von S nach $\bar{S}$ laufen und der, die von $\bar{S}$ nach S laufen. Da der Fluss von $\bar{S}$ nach S zurück fließender Fluss ist, muss er abgezogen werden.

$$F = \underbrace{\sum_{(i,j) \in (S,\bar{S})} x_{ij}}_{maximal} - \underbrace{\sum_{(i,j) \in (\bar{S},S)} x_{ij}}_{\geq 0}$$

$\Rightarrow F \leq u[S, \bar{S}]$

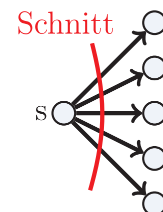
Dies gilt für alle (s, t)-Schnitte. Insbesondere für den mit minimaler Kapazität.

$u[S, \bar{S}] \Rightarrow$ Der Wert eines maximalen Flusses ist nicht größer als die Kapazität eines minimalen Schnittes.

$F_{max} \leq MinCut$ (Schwache Variante des MaxFlow/MinCut-Theorem)

Obere Schranke der Kapazität des (s,t)-Schnittes

Sei $\mathcal{U}$ = maximale Kapazität aller Kanten $(max\{u_{ij} | (i, j) \in E\})$
Kapazität eines Schnittes $\leq (n - 1) \cdot \mathcal{U}$
$\Rightarrow F_{max} \leq n \cdot \mathcal{U}$



5.3 Algorithmen auf G(x)

Es gibt zwei Möglichkeiten auf $G(x)$ zu arbeiten. Explizit darauf zu arbeiten, bedeutet, dass intern ein zweiter Graph aufgebaut wird. Dieser Graph besitzt sowohl die Originalkanten, als auch die Gegenkanten, solange es keine Null-Kanten sind.

Bei diesen Graphen kann z.B. DFS oder BFS zur Pfadsuche eingesetzt werden. Asymptotisch schneller ist es, wenn man implizit arbeitet, also auf dem Originalgraphen. Dabei müssen bei einer Pfaderhöhung Kanten eingefügt bzw. gelöscht werden. Im Gegensatz zum expliziten Restnetzwerk müssen hier nicht nur die ausgehenden, sondern auch die eingehenden Kanten betrachtet werden.

cap: Kapazitäten (u_{ij})
flow: Aktueller Flusswert (x_{ij})

```
1  forall_out_edges(e,v){
2        r = cap[e] - flow[e];
3        if(r == 0)
4              continue; // Ausfiltern aller Kanten, die nicht in
                  G(x) sind
5        //Rumpf
6  }
7  forall_in_edges(e,v){
8        r = flow[e];
9        if(r == 0)
10             continue;
11       //Rumpf
12 }
```

Listing 5.1: Ausfiltern der Kanten

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     $\delta = min\{r_{ij} | (i, j) \in P\}$
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

```
void AUGMENT(const graph& G, node s, node t, const
   node_array<edge>& PRED, const edge_array<int>& cap,
   edge_array<int>& flow){
     int delta = MAXINT; // Restkapazität von P
     node v = t;
     while(v != s){
          int r;
          edge e = PRED[v];
          if(v == G.source(e)){ //Rückwärtskante
               r = flow[e];
               v = G.target(e);
          }
          else{ // Vorwärtskante
               r = cap[e] - flow[e];
               v = G.source(e);
          }
          if(r < delta)
               delta = r; // min
     }
     // Eigentliche Flusserhöhung
     v = t;
     while(v != s){
          edge e = PRED[v];
          if(v == G.source(e)){ //Rückwärtskante
               flow[e] = flow[e] - delta;
               v = G.target(e);
          }
          else{ // Vorwärtskante
               flow[e] = flow[e] + delta;
               v = G.source(e);
          }
     }
}
```

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-

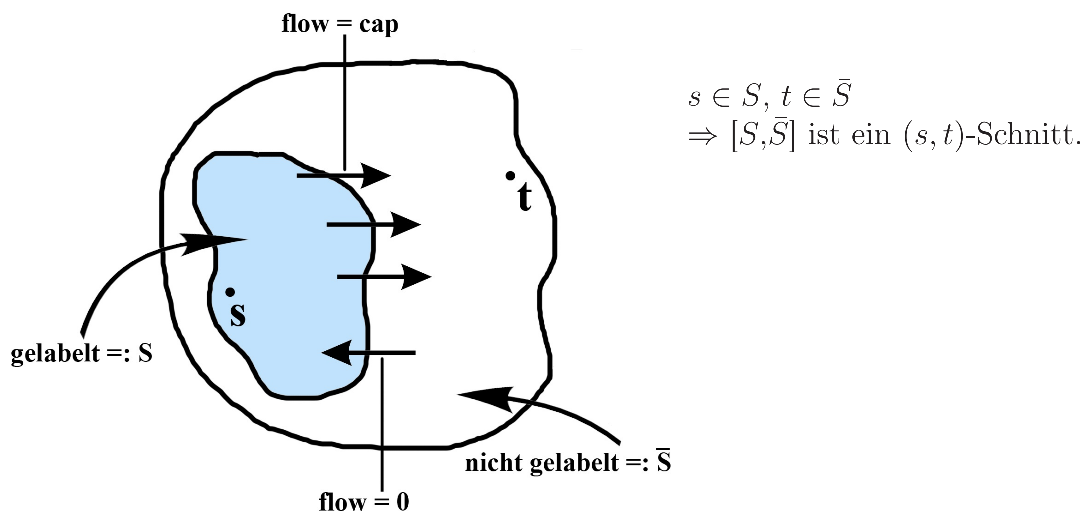
ge L hinzugefügt wird, wird s zu Beginn immer gelabelt und in L eingefügt. In der inneren Schleife wird zuerst ein beliebiger Knoten u aus der Menge L entnommen. Von diesem Knoten u aus werden nun sowohl die ausgehenden, als auch die eingehenden Kanten betrachtet. Dabei werden diejenigen Kanten, die 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: $u_ij = 0$ (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: $F = \sum_{(i,j)\in(S,\bar{S})} x_{ij} - \sum_{(i,j)\in(\bar{S},S)} x_{ij}$
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)

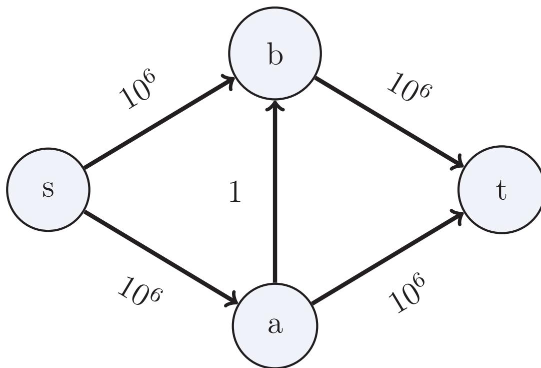
Beobachtung:
Höchstens $n - 1$ Kanten verlaufen von s nach t

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

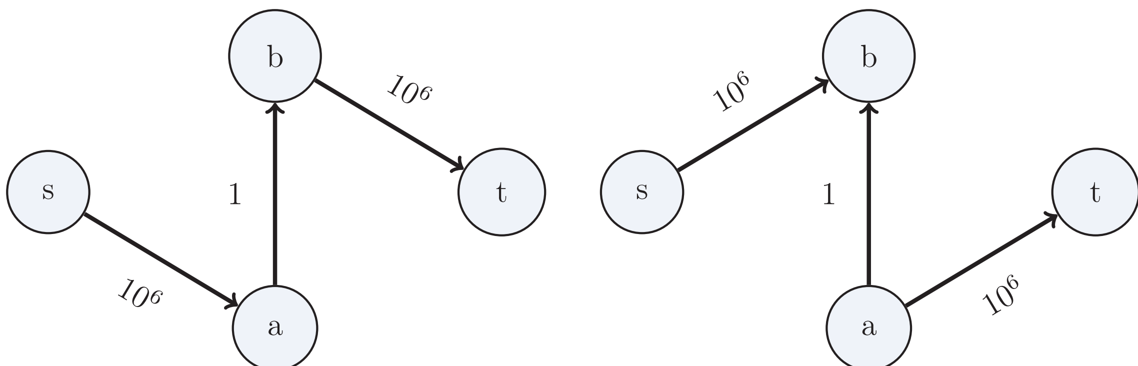
In jeder Iteration (AUGMENT) erhöht der Algorithmus den Flusswert um mindestens 1.
Daraus folgt, dass maximal F_{max} Iterationen benötigt werden, also maximal $n \cdot \mathcal{U}$ viele Iterationen.
Eine Iteration (Labeling) kostet Zeit $\mathcal{O}(n+m)$ (siehe DFS, BFS ...)
$\Rightarrow$ Gesamtlaufzeit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

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. ($\rightarrow$ Capacity Scaling)
2. Suche nach kürzesten erhöhenden Pfaden (# Kanten) ($\rightarrow$ 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.
$\rightarrow$ 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.
$\leq 2 \cdot m$ 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) $\sum\limits_{\substack{\text{für ein-}\\ \text{gehende}\\ \text{Kanten}}} x_{ji} - \sum\limits_{\substack{\text{für aus-}\\ \text{gehende}\\ \text{Kanten}}} x_{ik} \geq 0 \;\forall\, i \in V \setminus \{s,t\}$

   (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) $d(i) \leq d(j) + 1 \; \forall \; (i, j) \in G(x)$

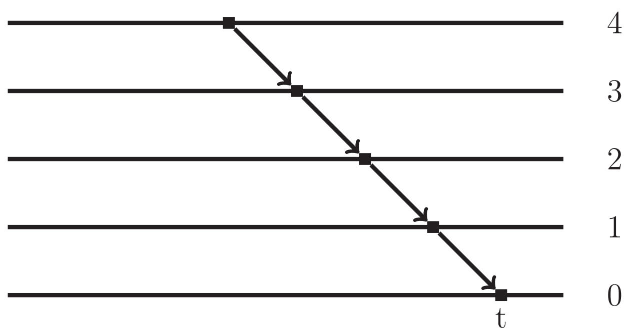
Beobachtung:

i) d ist untere Schranke für exakte Distanzwerte.

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

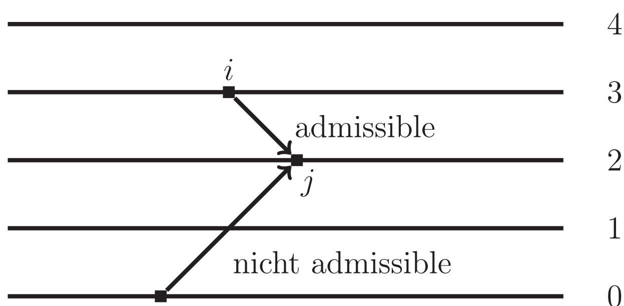
Intention:

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



Definition admissible

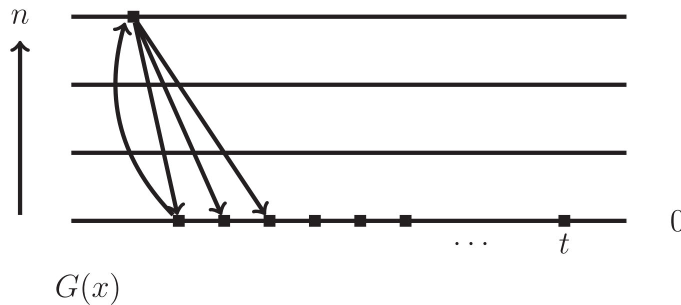
Eine Kante (i,j) in $G(x)$ heißt admissible, wenn $d(i) = d(j) + 1$ dh. der Höhenunterschied genau 1 beträgt.



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

```
1  foreach v ∈ V do
2  |   d(v) → 0
3  |   e_v → 0
4  end
5  foreach (u, v) ∈ E do
6  |   x_uv → 0
7  end
8  foreach j ∈ V mit (s, j) ∈ E do
9  |   x_sj ← u_sj     // volle Kapazität
10 |   e_j ← e_j + x_sj
11 end
12 d(s) ← n
13 while ∃ aktive Knoten (dh. e(i) > 0) do
14 |   Wähle einen aktiven Knoten i
15 |   PUSH/RELABEL(i)
16 end
```

Algorithmus 5 : Push/Relabel(i)

```
1  if ∃ admissible Edge (i, j) in G(x) then
2  |   Wähle eine solche Kante (i, j)
3  |   δ ← min{e(i), r_ij}     Überschuss, Restkapazität
4  |   x_ij ← x_ij + δ
5  |   e(i) ← e(i) − δ
6  |   e(j) ← e(j) + δ
7  end
8  else
9  |   d(i) ← min{d(j)|(i, j) in G(x)} + 1
10 end
```

Laufzeit:
$\mathcal{O}(\#\text{ Pushes} + \#\text{ Relabels})$

Lemma 7
Falls $e(i) > 0$ (i aktiv), dann existiert ein Pfad von s nach i in $G(x)$.

Lemma 8
Für alle aktiven Knoten $i \in V$ gilt $d(i) < 2n$.
(Folgt aus Lemma 7)

Lemma 9
Es werden maximal $2n^2$ Relabels durchgeführt.

Beweis
Für jeden einzelnen Knoten werden maximal $2n$ Relabels durchgeführt.
(Folgt aus Lemma 8)

□

Lemma 10
Es werden maximal nm saturierende Pushes durchgeführt.

Lemma 11
Es werden maximal n^2m nicht-saturierende Pushes durchgeführt.

Theorem 4

Der generische Preflow-Push-Algorithmus hat eine Laufzeit von $\mathcal{O}(n^2 \cdot m)$

A Anhang

A.1 Variablenerklärung

A.1.1 Kürzeste Wege

Variable	**Erklärung**
V	Knotenmenge
n	Anzahl an Knoten
E	Kantenmenge
m	Anzahl an Kanten
$G = (V, E)$	Gerichteter Graph
$cost : E \to \mathbb{R}$	Kostenfunktion
$cost(e)$	Kosten der Kante e
P	Pfad
$cost(P)$	Kosten des Pfades P
$dist(v, w)$	Wert des kürzesten Pfades von v nach w
s	Startknoten (Quelle, source)
t	Zielknoten (Senke, target)
$c((v, w))$	Kosten der Kante (v, w)
$DIST[v]$	Wert des kürzesten Pfades von s nach v, der bisher gefunden wurde
T	Baum, gebildet aus kürzesten Pfaden
U	Kandidatenmenge für verfeinerten Algorithmus
PQ	Priority Queue (Menge von Knoten mit dazugehörigen Prioritäten)
I	Information eines Eintrags in einer PQ (hier: *node*)
P	Priorität eines Eintrags in einer PQ (hier: $intDIST$-Wert)

A.1.2 Maximale Flüsse

Variable	**Erklärung**
V	Knotenmenge
E	Kantenmenge
$G = (V, E)$	Gerichteter Graph
s	Startknoten (Quelle, source)
t	Zielknoten (Senke, target)
P	Pfad
u_{ij}	Kapazität von (i, j)
x_{ij}	Flussfunktion
e	Kante in G
F	Flusswert
$G(x)$ Restnetzwerk	
r_{ij}	Restkapazität der Kante (i, j)
δ	Restkapazität eines Pfades
$MinCut$	Minimaler Schnitt
$[S, \bar{S}]$	(s, t)-Schnitt
$u[S, \bar{S}]$	Kapazität des (s, t)-Schnittes
$r[S, \bar{S}]$	Restapazität des (s, t)-Schnittes
$\mathcal{U}$	maximale Kapazität aller Kanten
cap	Kapazitäten (u_{ij})
$flow$	Aktueller Flusswert (x_{ij})
$pred[v]$	Kante über die v zum ersten Mal erreicht wurde
$MaxFlow$	Maximaler Fluss
Δ-Restnetzwerk $G(x, \Delta)$	$G(x)$ ohne alle Kanten mit $r_{ij} < \Delta$
$e(i)$	Überschuss oder Excess von i
d_{ij}	Distanzfunktion
$d(i)$	Höhe oder Level des Knoten i