Algorithmen und Datenstrukturen

Flüsse in Graphen, Min-Cut-Max-Flow

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Danksagung

Die nachfolgenden Präsentationen wurden mit ausdrücklicher Erlaubnis des Autors übernommen aus:

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 7,8,9) gehalten von Christian Scheideler an der TUM http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

Es wurden umfangreiche Veränderungen vorgenommen.

Betrachtete Arten von Netzwerken

- Internet
- Telefonnetz
- Autobahnen/Eisenbahnnetz
- Elektrizitätsnetz
- Öl-/Gaspipelines
- Kanalisation
- …

Netzwerke

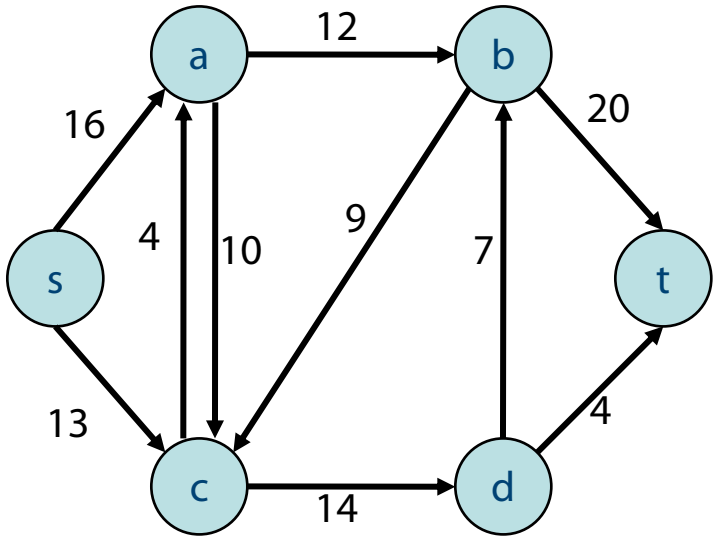
- Gegeben: Gerichteter Graph $G=(V, E)$
  - Kanten repräsentieren Flüsse von Material/Energie/Daten/…
    - Jede Kante hat eine maximale Kapazität, dargestellt durch (totale) Funktion $c: E \longrightarrow \mathbb{R}_+$
  - Knoten $s \in V$ als Quelle des Flusses
  - Knoten $t \in V$ als Senke des Flusses

- Ein Netzwerk ist ein Tupel (G, c, s, t) mit $s \in V$ und $t \in V$
- Die Funktion c macht G zum gewichteten Graphen

- Für jede Kante eines Netzwerks ist die Größe des Flusses steuerbar, dargestellt durch (totale) Funktion $f: E \longrightarrow \mathbb{R}$

Problem des maximalen Flusses in Netzwerken

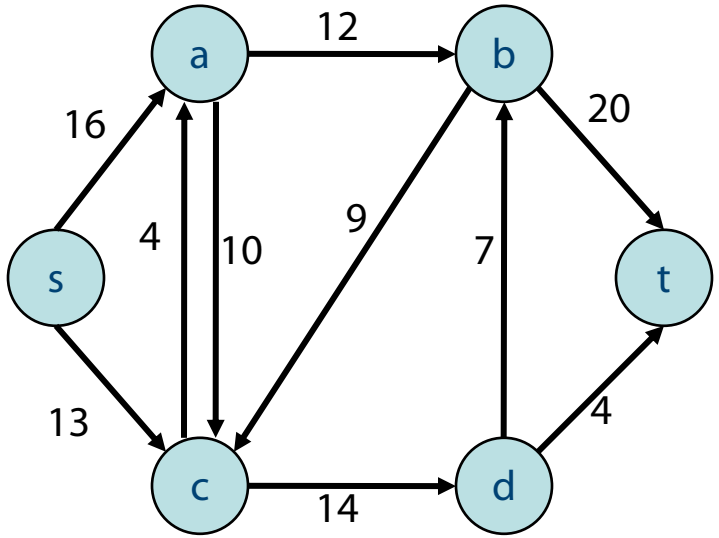
- Gegeben sei ein gerichteter gewichteter Graph
  - nicht-negative Gewichte
  - Gewichte repräsentieren Kapazität der Kanten (Funktion c)
- 2 ausgezeichnete Knoten s, t
  - s hat nur ausgehende Kanten
  - t hat nur eingehende Kanten
- Finde die maximale Anzahl von Einheiten , die von der Quelle zur Senke in diesem Graphen fließen kann
- Maximale Anzahl von Einheiten pro Einzelkante dargestellt durch Funktion $f_{max}: E \longrightarrow \mathbb{R}$



Jede Zahl steht für die Kapazität dieser Kante

Problem des maximalen Flusses in Netzwerken

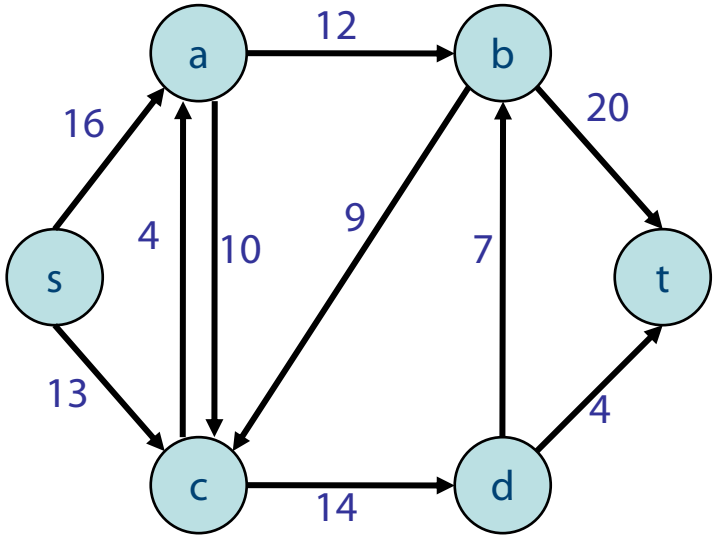
- Jede Kante könnte ein Wasserrohr darstellen
  - Von einer Quelle fließt Wasser zu einer Senke
  - Jedes Wasserrohr kann eine maximale Anzahl von Litern Wasser pro Sekunde transportieren



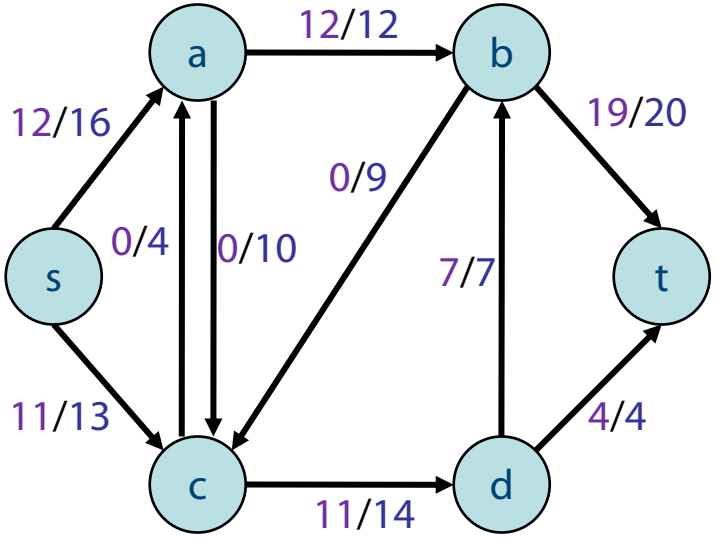
Jede Zahl steht für die Kapazität dieser Kante

- Wie viel Wasser pro Sekunde kann nun von s zu t maximal fließen?

Netzwerkfluss



Dieser Graph enthält die Kapazitäten jeder Kante im Graph (Beschriftung c(e))



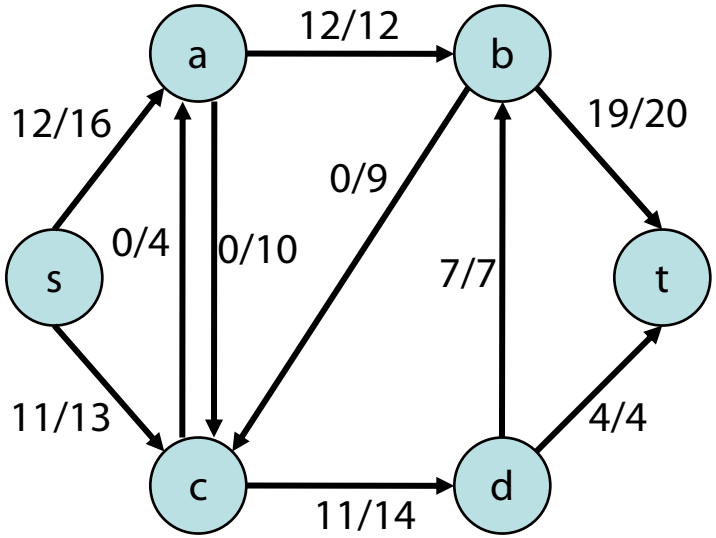
Dieser Graph enthält zusätzlich den Fluss im Graphen (Beschriftung f(e)/c(e)

- Der Fluss des Netzwerkes ist definiert als der Fluss von der Quelle s (oder in die Senke t)
- Im Beispiel oben ist der Netzwerkfluss 23

Netzwerkfluss

- Flusserhaltung:
  - Mit Ausnahmen der Quelle s und Senke t ist der Fluss, der in einen Knoten hineinfließt, genauso groß wie der Fluss, der aus diesem Knoten herausfließt

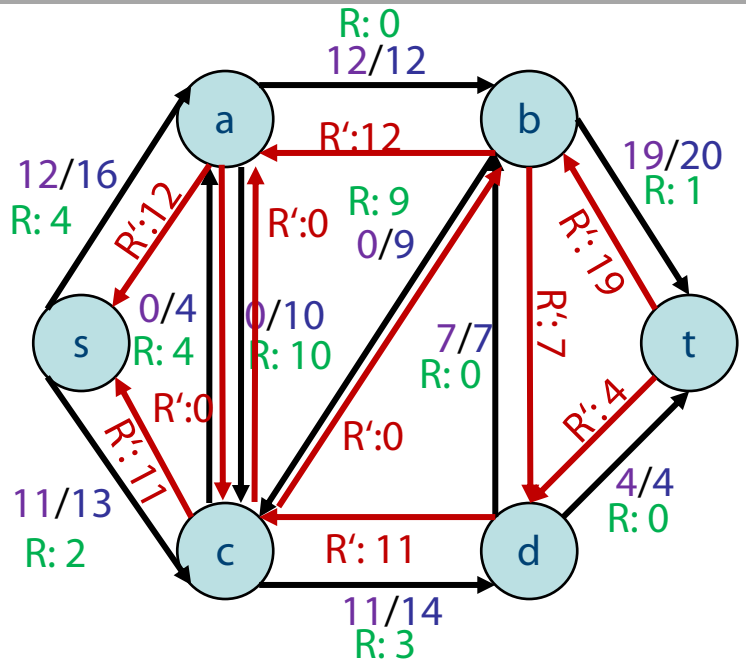
- Beachtung maximaler Kapazitäten:
  - Jeder Fluss in einer Kante muss kleiner oder gleich der Kapazität dieser Kante sein



Fluss / Kapazität im Graph

Netzwerkfluss

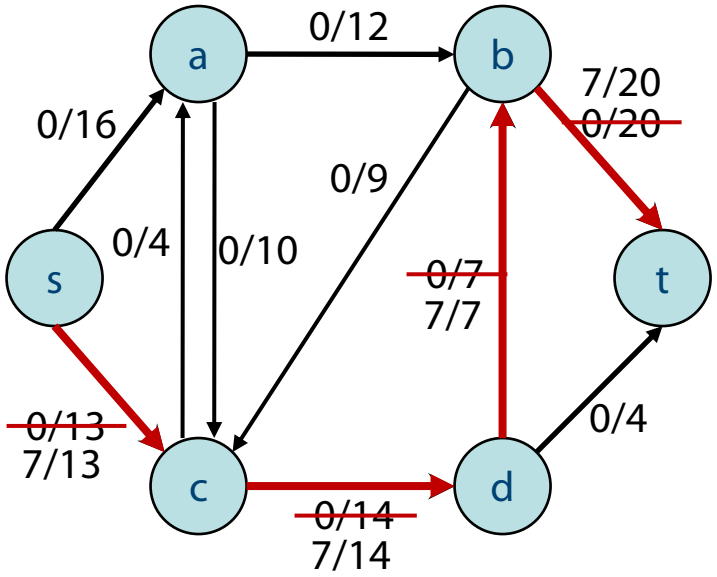
- Restkapazität einer Kante
  - Unbenutzte Kapazität jeder Kante
  - Zu Beginn ist der Fluss 0 und damit ist die Restkapazität genau so groß wie die Kapazität
  - Existiert ein Fluss, so kann der Fluss auch wieder reduziert werden, dies ist wie eine Restkapazität in die entgegengesetzte Richtung
- Restkapazität eines Pfades
  - Minimale Restkapazität aller Kanten entlang des Pfades
- Flusserhöhender Pfad
  - Pfad von der Quelle zur Senke mit Restkapazität größer als 0
  - Kann auch „Restkapazitäten in die entgegengesetzte Richtung“ beinhalten



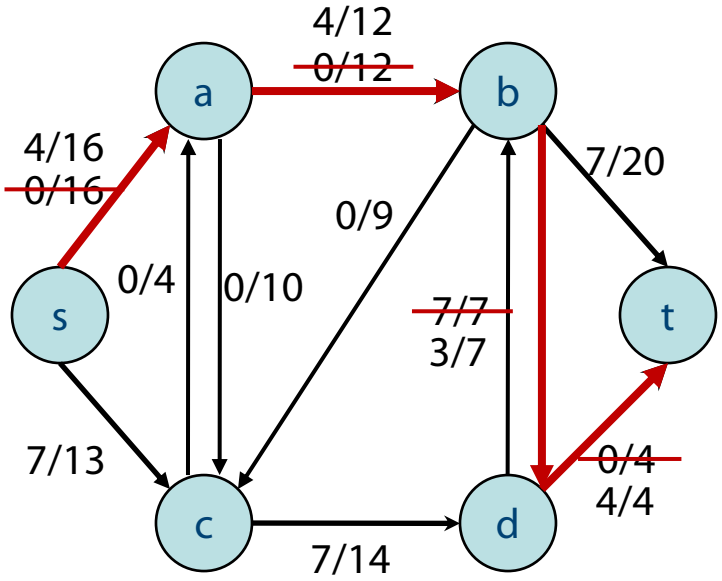
Fluss / Kapazität im Graph
Restkapazität R: Kapazität – Fluss
Restkapazität R' in die entgegengesetzte Richtung: Fluss

Beispiel für flusserhöhende Pfade

Flusserhöhender Pfad nur mit „normalen“ Restkapazitäten



Flusserhöhender Pfad auch mit Restkapazitäten in die entgegengesetzte Richtung



Restkapazität ist auch für Pfade entsprechend definiert

Ford-Fulkerson-Algorithmus (Skizze)

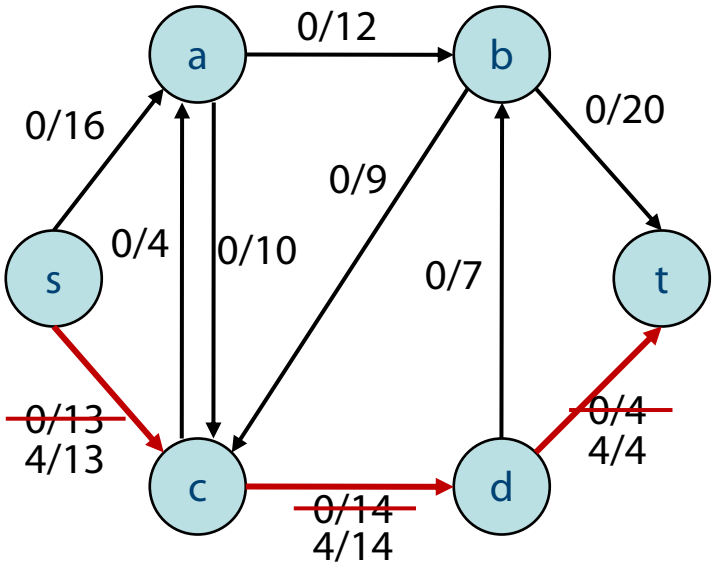
```
procedure Ford-Fulkerson (G, s, t, f):
   // Sei s Quelle und t Ziel in G=(V, E) mit s, t ∈ V
   for (u, v) ∈ E do f(u,v) := 0
   while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
     // Betrachtungsreihenfolge der Pfade bleibt offen
      for (u_i, v_i) ∈ p do
          if forward-edge((u_i, v_i), G)
             then f(u_i, v_i) := f(u_i, v_i) + rest-capacity(p)
             else f(u_i, v_i) := f(u_i, v_i) - rest-capacity(p)
   return f(s, t)

function forward-edge((u v), (V, E))
      return (u, v) ∈ E
```

Nichtdeterministischer Algorithmus

Ford, L. R.; Fulkerson, D. R. "Maximal flow through a network". Canadian Journal of Mathematics 8: 399. **1956**

Ford-Fulkerson Algo – Beispieldurchlauf

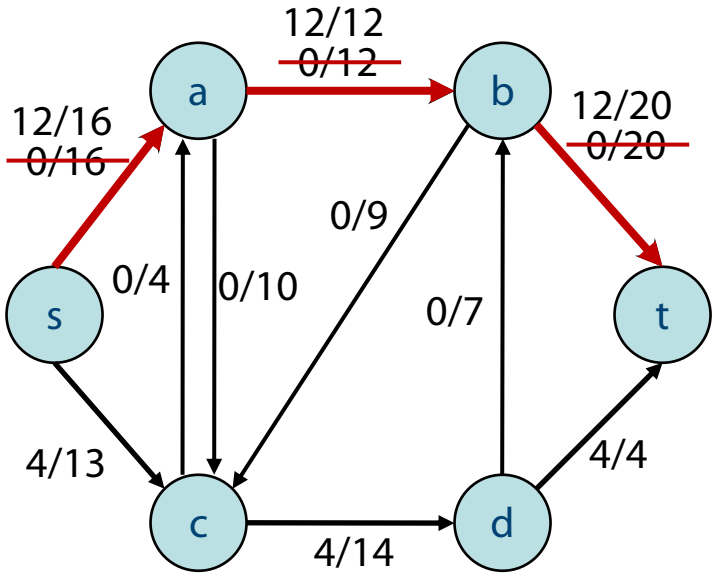
while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
  Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Wähle flusserhöhenden Pfad, z.B. s, c, d, t
- Restkapazität dieses Pfades ist 4

Ford-Fulkerson Algo – Beispieldurchlauf

while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Wähle anderen zunehmenden Pfad, z.B. s, a, b, t
- Restkapazität dieses Pfades ist 12

Pfadbestimmung z.B. mit Tiefensuche

Ford-Fulkerson-Algorithmus mit DFS

```
procedure Ford-Fulkerson (G, s, t, f):
    // Sei s Quelle und t Ziel in G=(V, E) mit s,t∈V
    // f wird modifiziert
    (V, E) := G
    for (u, v) ∈ E do f(u,v) := 0
    while true do
        p := FF-DFS(G, s)
        if p = ⊥ then return f(s , t)
        for (u_i, v_i) ∈ p do
            if forward-edge((u_i, v_i), G)
                then f(u_i, v_i) := f(u_i, v_i) + rest-capacity(p)
                else f(u_i, v_i) := f(u_i, v_i) – rest-capacity(p)
```

Ford, L. R.; Fulkerson, D. R. "Maximal flow through a network". Canadian Journal of Mathematics 8: 399. **1956**

Tiefensuche – Schema: FF-DFS

```
Function FF-DFS((V,E), s) :
    unmark all nodes
    init()
    DFS(((V,E), s,s)    // s: Sourceknoten
    return ⊥

Procedure DFS(((V,E), u,v: Node)  // u: Vater von v
    if not exists (v,w)∈E and flow-augmenting-path( path(v) ) then // v=t
        return-from FF-DFS path(v)
    for (v,w)∈E do
        if w is not marked then
            mark w with predecessor v //forward-edge((v, w), …) returns true
            DFS((V,E), v, w)
    for (w,v) ∈ E do
        if w is not marked then
            invmark w with predecessor v  //forward-edge((v, w), …) returns false
            DFS((V,E), v,w)
    backtrack(u,v)

backtrack(u,v) : unmark(v)
```

Viele viele Verbesserungsmöglichkeiten denkbar

Weitere Code-Muster und Prozeduren

- Variablen
  - parents, invparents : Array[1..n] of NodeId
- unmark all nodes

  for i from 1 to n do parents[i] := ⊥; invparents[i] := ⊥
- init()
  - -
- w is not marked

  parents[w] = ⊥ ∧ invparents[w] = ⊥
- mark w with predecessor v
  - parents[w] := v
- invmark w with predecessor v
  - invparents[w] := v
- unmark(v)
  - parents[v] := ⊥; invparents[v] := ⊥
- path(v)

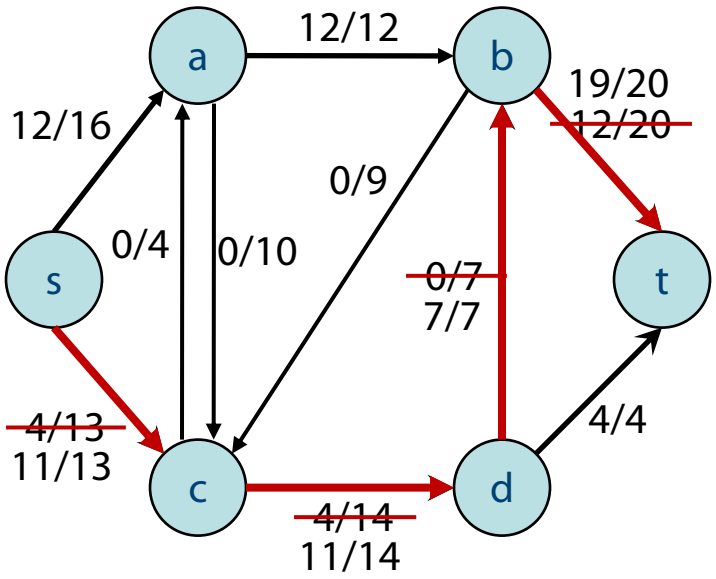
  // Verwende parents- und invparents-Feld, um Pfad zu konstruieren

  // Speichere Restkapazität des Pfades und Kantenrichtung
- rest-capacity(p)

  // Getter

Ford-Fulkerson Algo – Beispieldurchlauf

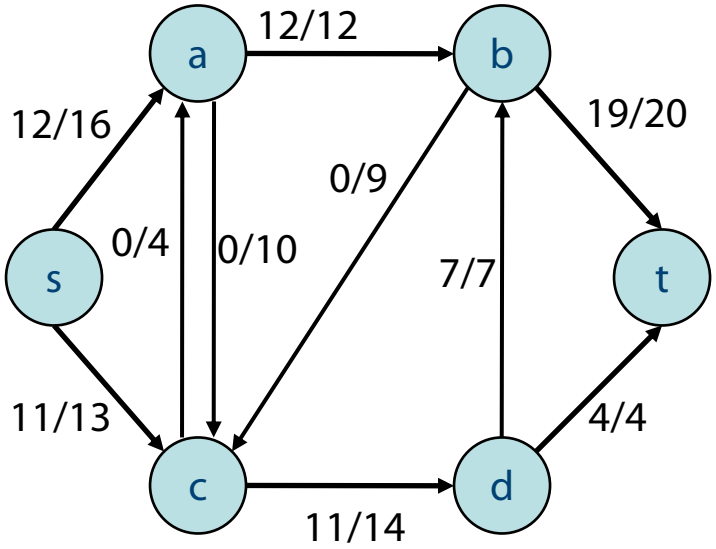
while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Wähle anderen zunehmenden Pfad, z.B. s, c, d, b, t
- Restkapazität dieses Pfades ist 7

Ford-Fulkerson Algo – Beispieldurchlauf

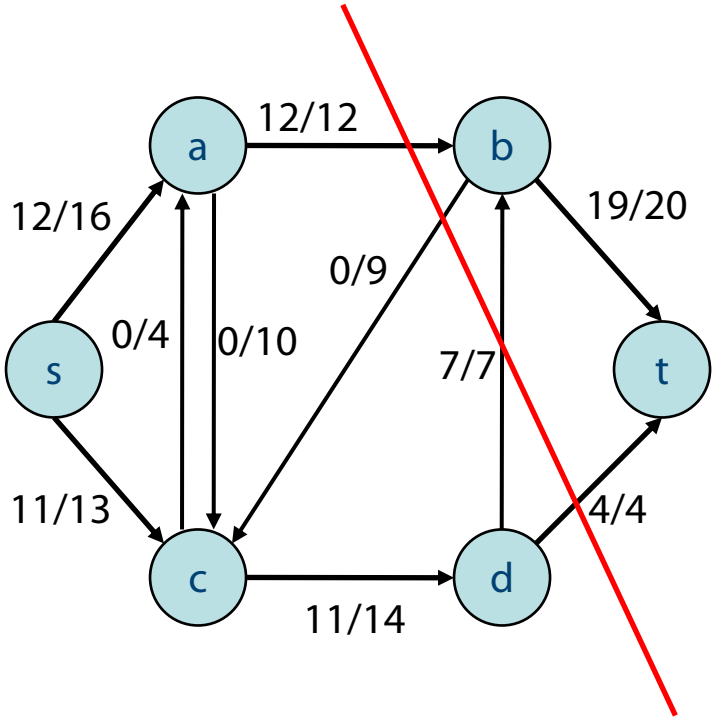
while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Gibt es weitere flusserhöhende Pfade?
  Nein!
  Fertig
- Maximaler Fluss:
  19+4 = 23 (bzw. 11+12)

Ford-Fulkerson Algo – Beispieldurchlauf

while ∃p ∈ paths(s, t, G) : flow-augmenting-path(p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Gibt es weitere flusserhöhende Pfade?
  Nein!
  Fertig
- Maximaler Fluss:
  19+4 = 23 (bzw. 11+12)

Analyse des Algorithmus von Ford/Fulkerson

- Ein Schnitt in $N=((V, E), c, s, t)$ ist ein disjunkte Zerlegung von V in Mengen $S \subseteq V$ und $T \subseteq V$ mit $s \in S$, $t \in T$.
- Die Kapazität des Schnittes ist $c(S, T) = \sum_{e \in E \cap (S \times T)} c(e)$
- Die Kapazität eines minimalem Schnittes ist
  $c_{min} = \min_{(S, T) \text{ Schnitt in } N} c(S, T)$
- Der Fluss eines Schnittes ist
  $f((S, T)) = \sum_{e \in E \cap (S \times T)} f(e) - \sum_{e \in E \cap (T \times S)} f(e)$
- Mit f_{max} bezeichnen wir den Wert eines maximalen Flusses

Max Flow/Min Cut-Theorem

- In jedem Netzwerk N=(G, c, s, t) gilt: Der Wert eines jeden Flusses ist kleiner oder gleich der Kapazität eines jeden Schnittes. Insbesondere gilt: $f_{max} \leq c_{min}$.
- Sei f der vom F.F.-Algo für N=(G, c, s, t) berechnete Fluss. Dann gibt es einen Schnitt (S,T) in N mit f(G) = c(S,T).

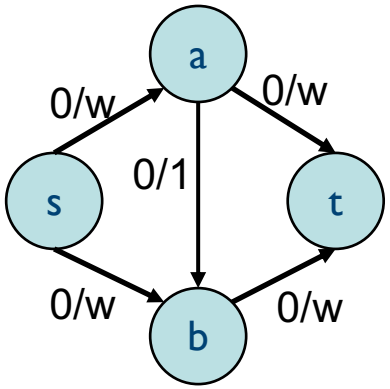
- Satz: (Max Flow-Min Cut Theorem; Satz von Ford/Fulkerson)
  Der Algorithmus von Ford/Fulkerson berechnet einen maximalen Fluss. In jedem Netzwerk gilt $f_{max} = c_{min}$.
  (ohne formalen Beweis)

Der Wert eines maximalen Flusses ist gleich der Kapazität eines minimalen Schnittes.

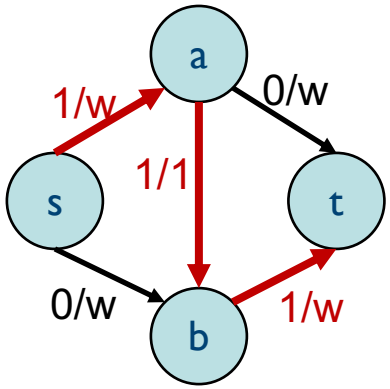
Ford-Fulkerson Algorithmus – Analyse

- Finden eines flusserhöhenden Pfades z.B. mit einer Tiefensuche: $O(n + m)$
- Aber: Pfade können über Tiefensuche in einer ungünstigen Reihenfolge betrachtet werden

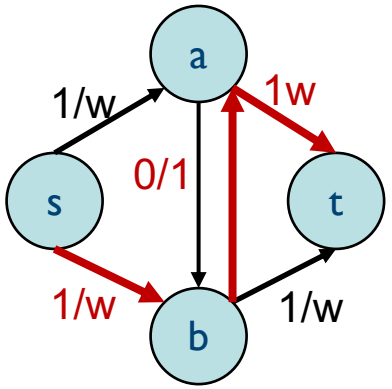
Schlechte Abfolge von zunehmenden Pfaden



1. flusserhöhender Pfad



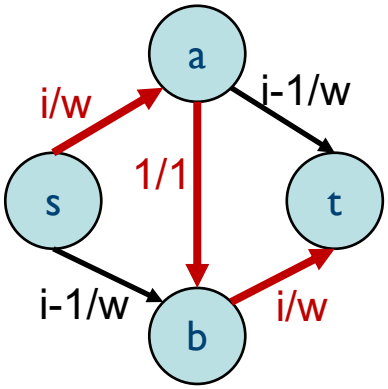
2. flusserhöhender Pfad



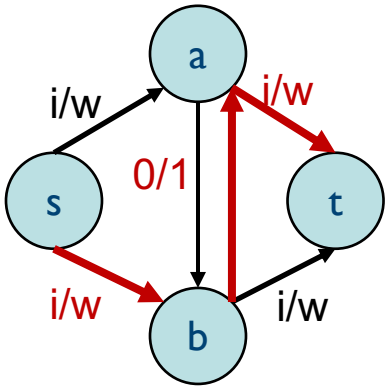
…

Fertig nach 2w flusserhöhenden Pfaden, obwohl der Algo auch schon mit 2 günstigen flusserhöhenden Pfaden fertig sein könnte!

2i-1. flusserh. Pfad

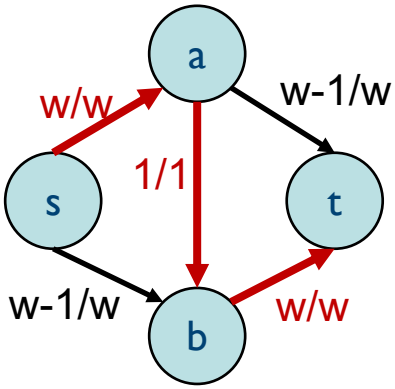


2i. flusserh. Pfad

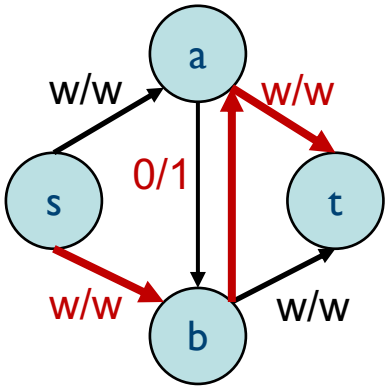


…

2w-1. flusserh. Pfad



2w. flusserh. Pfad



Ford-Fulkerson Algorithmus – Analyse

Damit ergibt sich mit f_{max}, dem maximalen Fluss von G, und der Verwendung von Tiefensuche als totale Laufzeit:

$$f_{max}(G) \cdot O(n+m)$$

Da für die betrachteten Gs gilt $m \gg n$ gilt, bekommen wir:

$$T_{\text{Ford-Fulkerson}}(G) \in f_{max}(G) \cdot O(m)$$

Man beachte: Wenn wir die Zahl f_{max} binär codiert als k-stelligen Bitvektor aus $\{0,1\}^k$ sehen, gibt es 2^k viele Erhöhungen von 0^k um 1, bis Wert f_{max} erreicht

F.F. ist also in diesem Sinne exponentiell in der Länge k

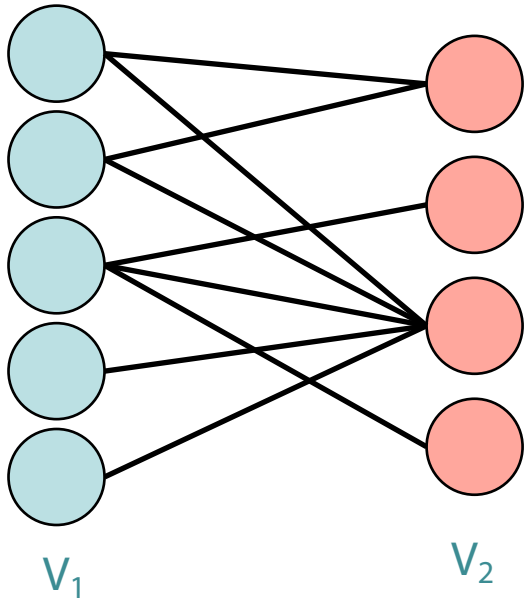
Edmonds-Karp Algorithmus

- Variation des Ford-Fulkerson Algorithmus durch Wählen von günstigen flusserhöhenden Pfaden
  - Wähle als nächstes den flusserhöhenden Pfad mit einer minimalen Anzahl von Kanten
    - durch Breitensuche ermittelbar
- Maximale Anzahl von betrachteten flusserhöhenden Pfaden, und damit Schleifendurchläufen: $n \cdot m$
  - Ohne Beweis
- $T_{\text{Edmonds-Karp}}(n,m) \in O(n \cdot m^2)$
  - Berechnung des maximalen Flusses im Beispiel mit 2 flusserhöhenden Pfaden

Jack Edmonds, Richard M. Karp: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. In: J. ACM. 19, Nr. 2, S. 248-264, **1972**

Anwendung: Maximale bipartite Matchings

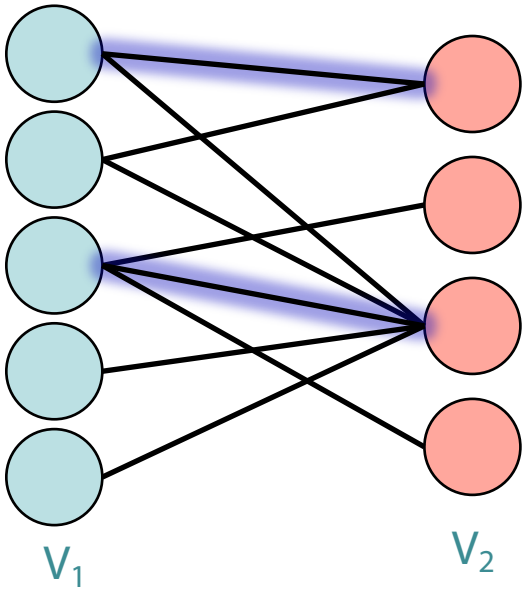
- Bipartite Graphen sind Graphen $G=(V, E)$ in denen die Knotenmenge V in zwei disjunkte Knotenmengen V_1 und V_2 aufgeteilt werden können $(V = V_1 \cup V_2)$, so dass
  $\forall (u, v) \in E: (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$
- Beispiel eines bipartiten Graphen:
  - Knoten aus V_1 repräsentieren ausgebildete Arbeiter und
  - Knoten aus V_2 repräsentieren Aufgaben,
  - Kanten verbinden die Aufgaben mit den Arbeitern, die sie (bzgl. ihrer Ausbildung) ausführen können



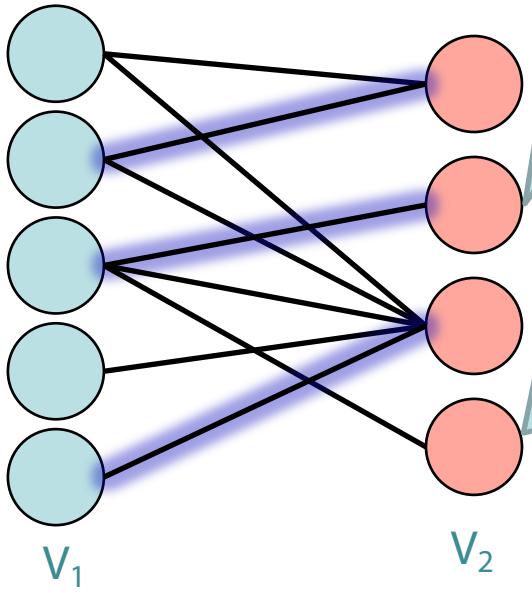
Bipartites Matching

- Finde $E' \subseteq E$, so dass $\forall v \in V$: degree(v)≤1 bezüglich E‘
  - 1 Arbeiter kann zur selben Zeit nur 1 Aufgabe erledigen und 1 Aufgabe braucht nur max. von einem Arbeiter bearbeitet zu werden
- Maximales bipartites Matching: $|E'|$ maximal
  - maximale Aufteilung der Aufgaben
    - so wenig Aufgaben wie möglich bleiben liegen und
    - so wenig Arbeiter wie möglich sind unbeschäftigt

Nicht maximal:



Maximal:

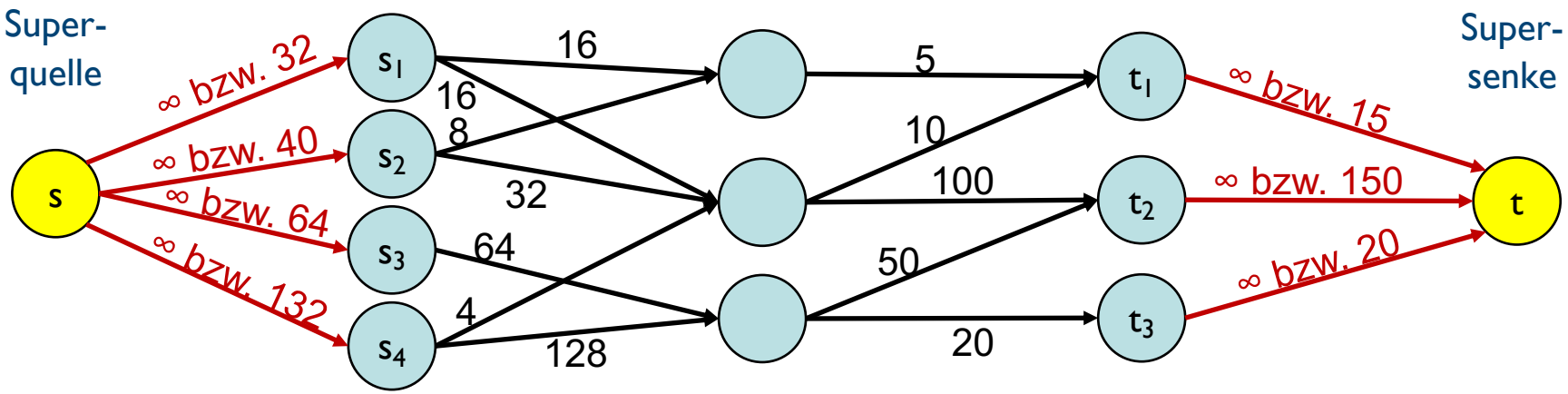


Diese Aufgaben können nur von ein und demselben Arbeiter erledigt werden, daher kein größeres Bipartites Matching möglich!

Mehrere Quellen und mehreren Senken

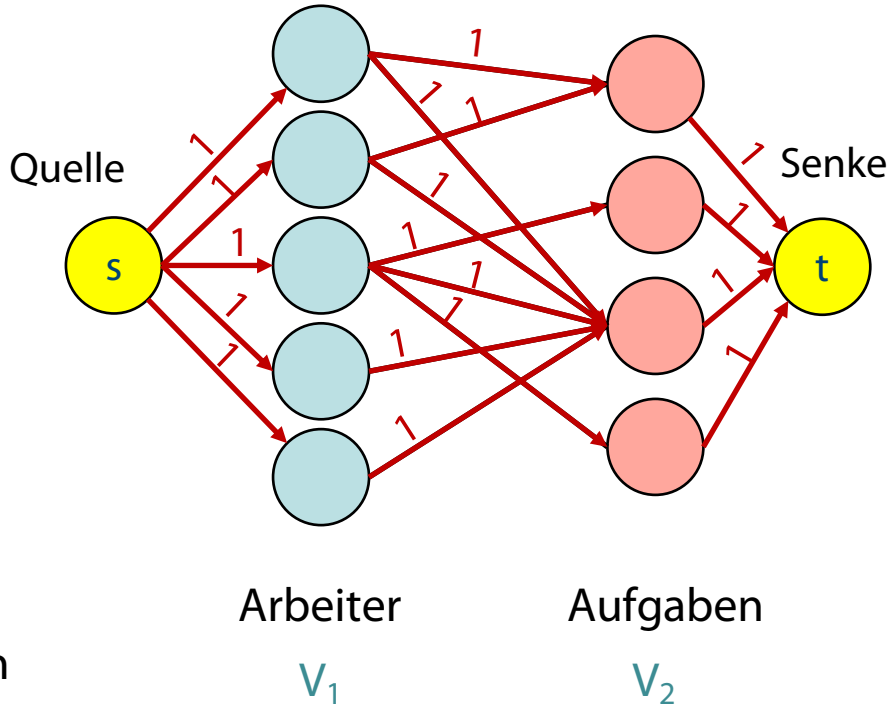
- Reduzierung auf maximalen Fluss in Netzwerk mit *einer* Quelle und *einer* Senke durch Einführung
  - einer Superquelle, die mit allen Quellen
  - einer Supersenke, die von allen Senken

  mit einer Kante mit unbeschränkter Kapazität verbunden ist
  - Anstatt Kanten mit unbeschränkter Kapazität kann man auch Kanten mit der Kapazität der entsprechenden Quelle bzw. Senke verwenden

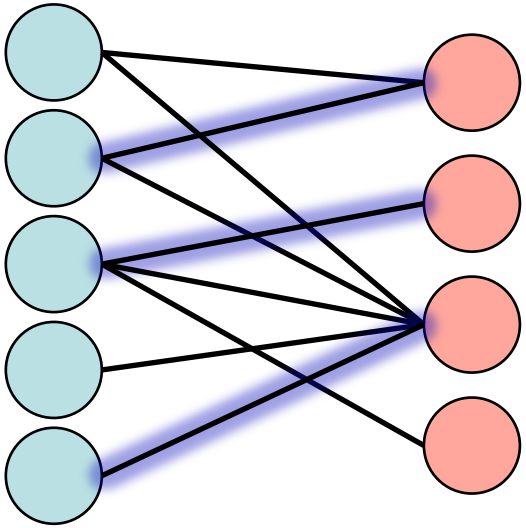


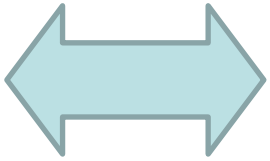
Lösung des maximalen Bipartiten Matchings

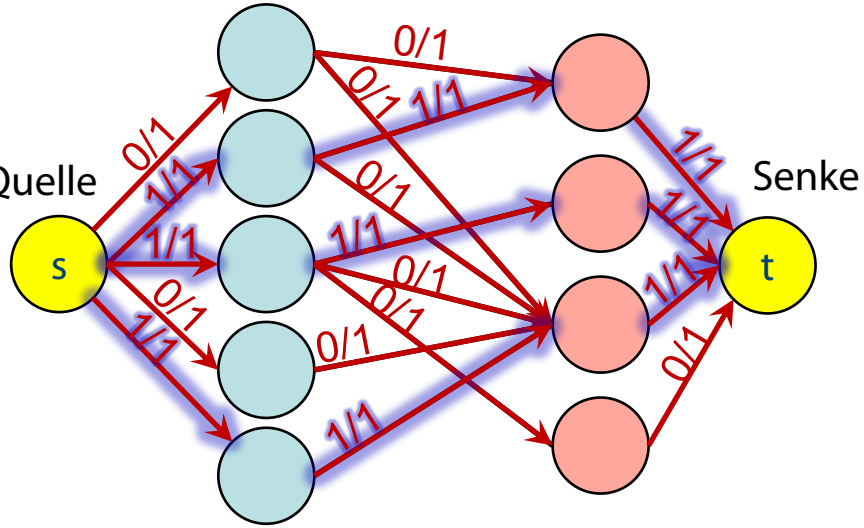
- Reduzierung auf das Problem des maximalen Flusses
  - Transformation des bipartiten Graphen auf einen Graphen für den Netzwerkfluss
    - Gerichtete Kanten von Knoten aus V_1 zu Knoten aus V_2 anstatt der ungerichteten Kanten des bipartiten Graphen
    - Einführung einer Quelle, die mit allen Knoten aus V_1 verbunden ist
    - Einführung einer Senke, die mit allen Knoten aus V_2 verbunden ist
    - Maximale Kapazität jeder Kante ist 1



Maximaler Fluss ⇔ Maximales bipartites Matching



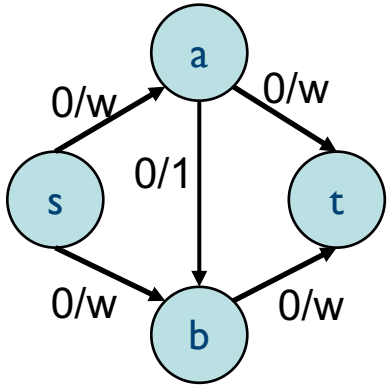




$G=(V_1 \cup V_2, E)$

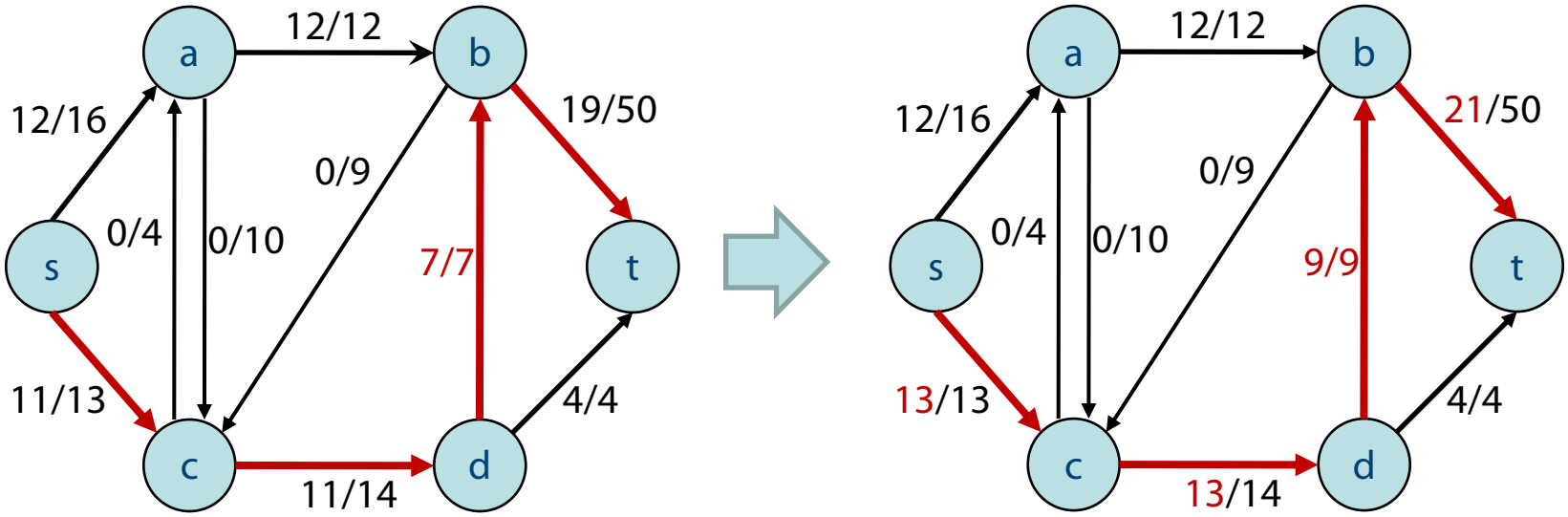
$T_{bipartite\text{-}match}(G) \in O(c \cdot (n+m))$

c bestimmt durch min der Kantenkostensumme vom Ausgang von s oder Eingang in t



Praktische Fragestellung

- Wie kann man durch Erhöhung der Kapazität an einer/wenigen Kanten den maximalen Fluss erhöhen?
  - Betrachte Pfade von der Quelle zu der Senke, deren Fluss die volle Kapazität einer Kante ausnutzen
  - Erhöhe die Kapazität der Kante(n), die die volle Kapazität ausnutzen, um das Minimum der Restkapazitäten der anderen Kanten des Pfades



Übersicht über Max-Flow-Algorithmen

(n=|V|, e=|E|, U=max{c(e) für alle e∈E})

Jahr	Autoren	Zeit gemessen in n, e, U	Zeit, wenn $e = \Omega(n^2)$
1969	Edmonds/Karp	$O(ne^2)$	$O(n^5)$
1970	Dinic	$O(n^2e)$	$O(n^4)$
1974	Karzanov	$O(n^3)$	$O(n^3)$
1977	Cherkasky	$O(n^2e^{1/2})$	$O(n^3)$
1978	Malhotra/Pramodh Kumar/ Maheshvari	$O(n^3)$	$O(n^3)$
1978	Galil	$O(n^{5/3}e^{2/3})$	$O(n^3)$
1978	Galil/Naamad sowie Shiloach	$O(ne\log^2 n)$	$O(n^3\log^2 n)$
1980	Sleator/Tarjan	$O(ne\log n)$	$O(n^3\log n)$
1982	Shiloach/Vishkin	$O(n^3)$	$O(n^3)$
1983	Gabow	$O(ne\log U)$	$O(n^3\log U)$
1984	Tarjan	$O(n^3)$	$O(n^3)$
1985	Goldberg	$O(n^3)$	$O(n^3)$
1986	Goldberg/Tarjan	$O(ne\log(n^2/e))$	$O(n^3)$
1986	Ahuja/Orlin	$O(ne + n^2\log U)$	$O(n^3 + n^2\log U)$
1989	Ahuja/Orlin/Tarjan	$O(ne + n^2\log U/\log\log U)$ $O(ne + n^2\log^{1/2} U)$ $O(ne\log(\frac{n}{e}\log^{1/2} U + 2))$	
1989	Cheriyan/Hagerup (rand.)	$O(ne + n^2\log^3 n)$	
	det. Version von Alon	$O(\min(ne\log n, ne + n^{8/3}\log n))$	
	det. Version von Tarjan	$O(\min(ne\log n, ne + n^2\log^2 n))$	
1990	Cheriyan/Hagerup/Mehlhorn	$O(n^3/\log n)$	
	rand.	$O(\min(ne\log n, ne + n^2\log^2 n, n^3/\log n))$	

Zusammenfassung

- Flüssen in Graphen
- Min-Cut-Max-Flow Algorithmen
- Zuordnungsprobleme