



07 Network Flow

CS216 Algorithm Design and Analysis (H)

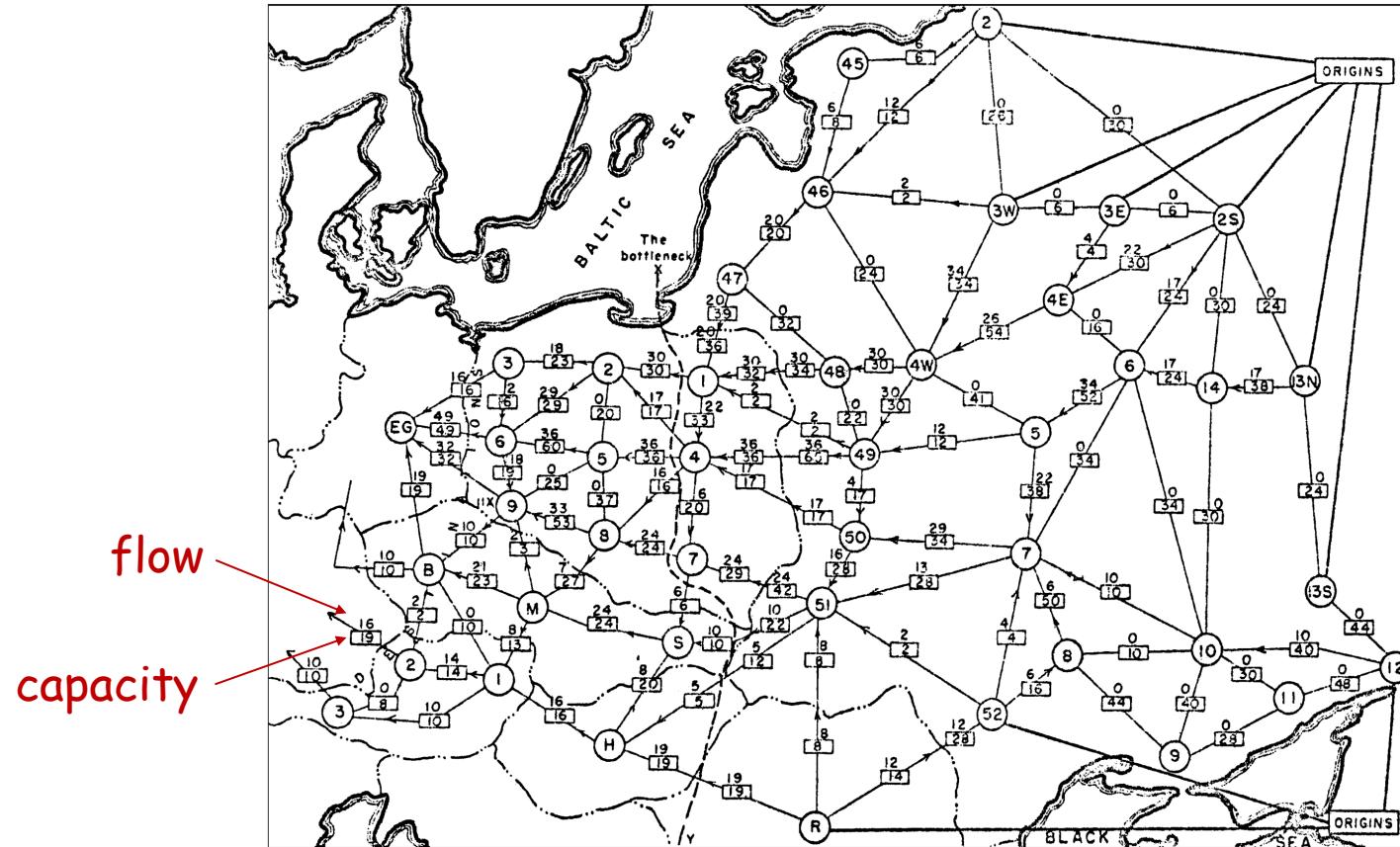
Instructor: Shan Chen

chens3@sustech.edu.cn



Maximum Flow Application (Tolstoi 1930s)

- Soviet Union goal. Maximize flow of supplies to Eastern Europe.

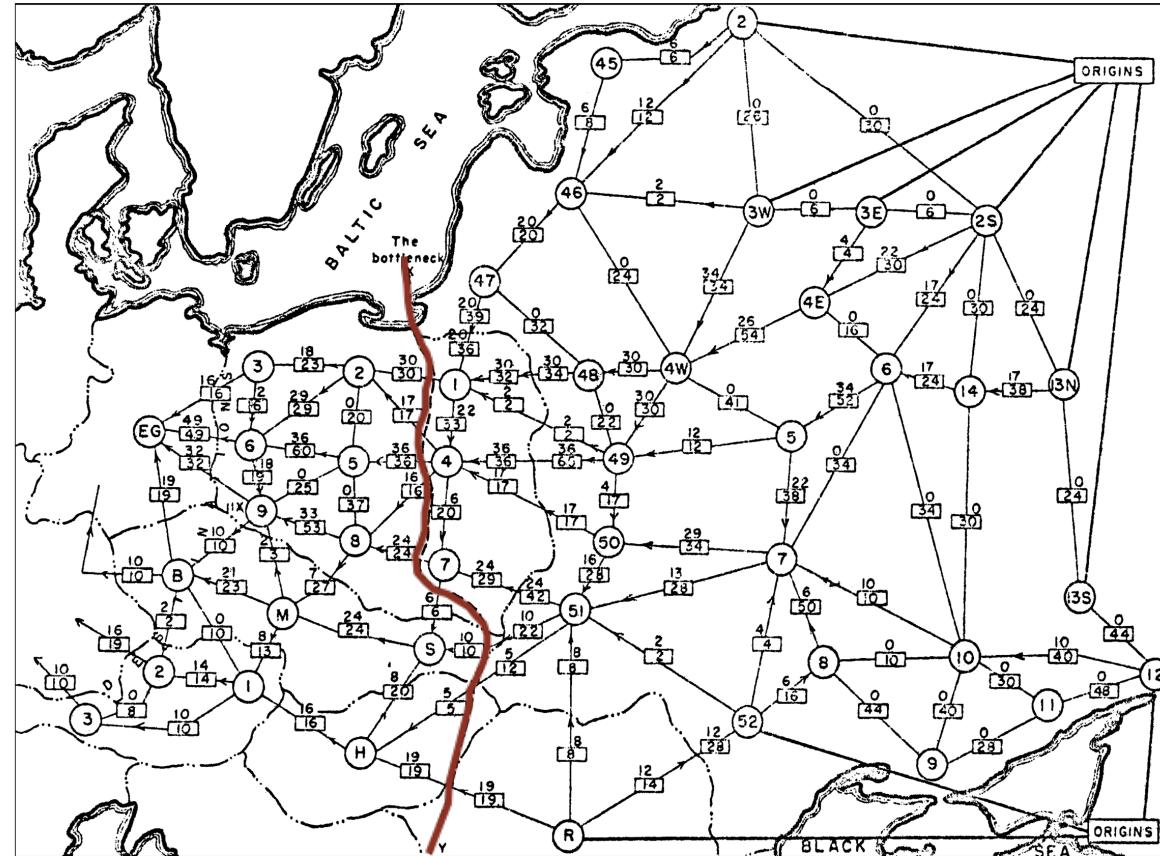


rail network connecting Soviet Union with Eastern European countries
(map declassified by Pentagon in 1999)



Minimum Cut Application (RAND 1950s)

- “Free World” goal. Cut supplies (if Cold War turns into real war).



rail network connecting Soviet Union with Eastern European countries
(map declassified by Pentagon in 1999)



Maximum Flow and Minimum Cut

- **Max-Flow and Min-Cut Problems:**

- Beautiful mathematical **duality**.
- Cornerstone problems in **combinatorial optimization**.

- **They are widely applicable models:**

- Data mining, open-pit mining, bipartite matching, network reliability, baseball elimination, image segmentation, network connectivity, Markov random fields, distributed computing, security of statistical data, egalitarian stable matching, network intrusion detection, multi-camera scene reconstruction, sensor placement for homeland security, etc.



we will learn some of the applications later

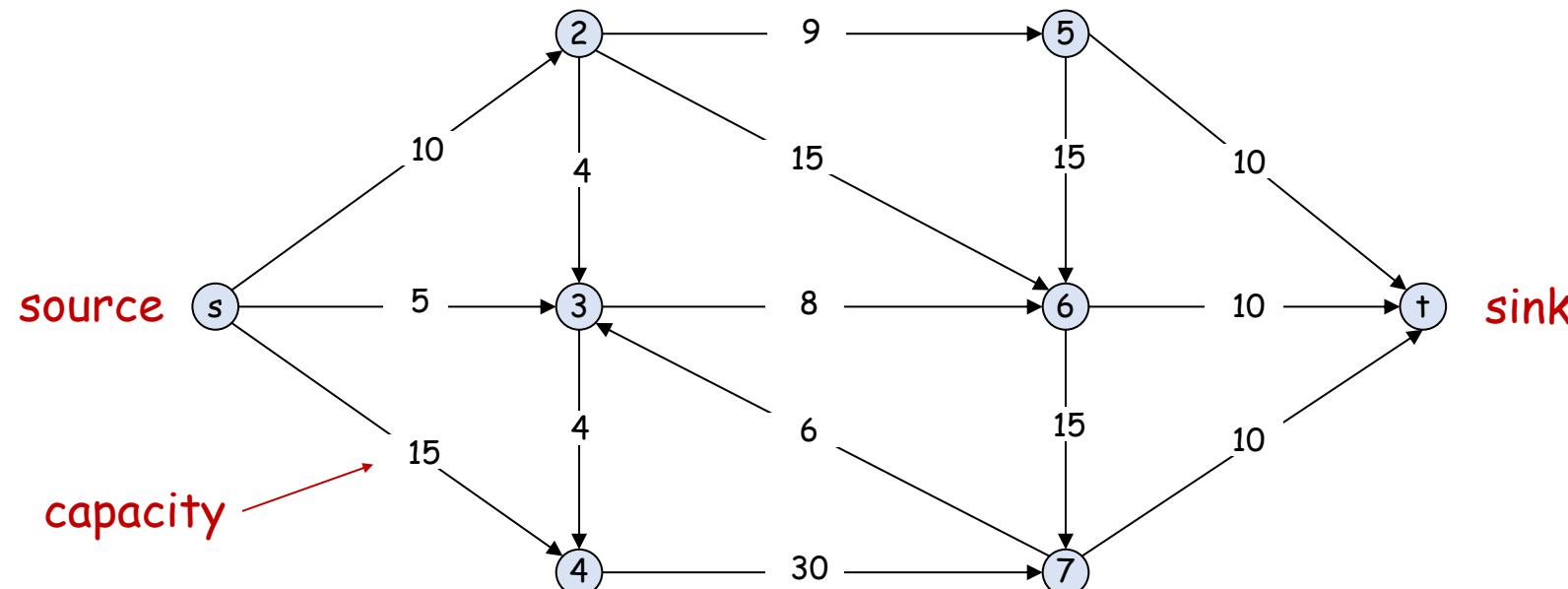


南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

1. Max Flow and Min Cut

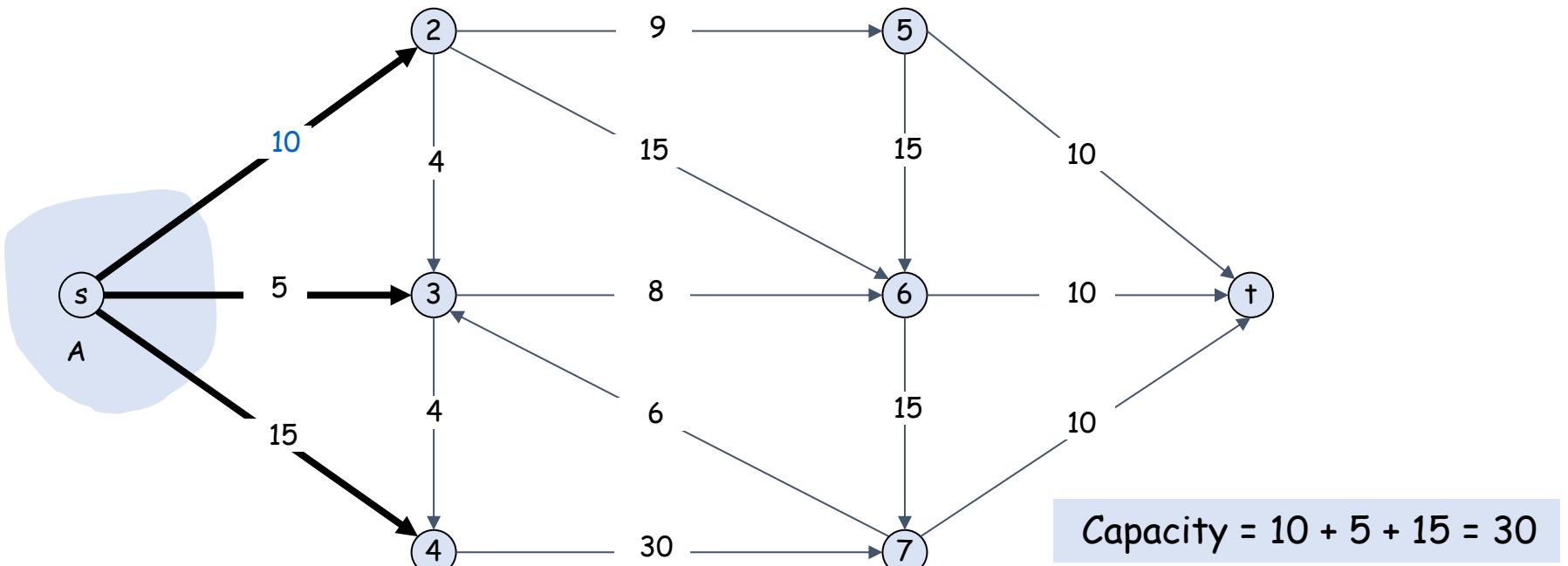
Flow Network

- **Def.** A flow network is a tuple $G = (V, E, s, t, c)$.
 - Directed graph $G = (V, E)$ with source s and sink t , and no parallel edges.
 - Capacity $c(e) \geq 0$ for each edge e .
- **Intuition.** Material flowing through a transportation network, originating from **source** and being sent to **sink**.
assume all nodes are reachable from s



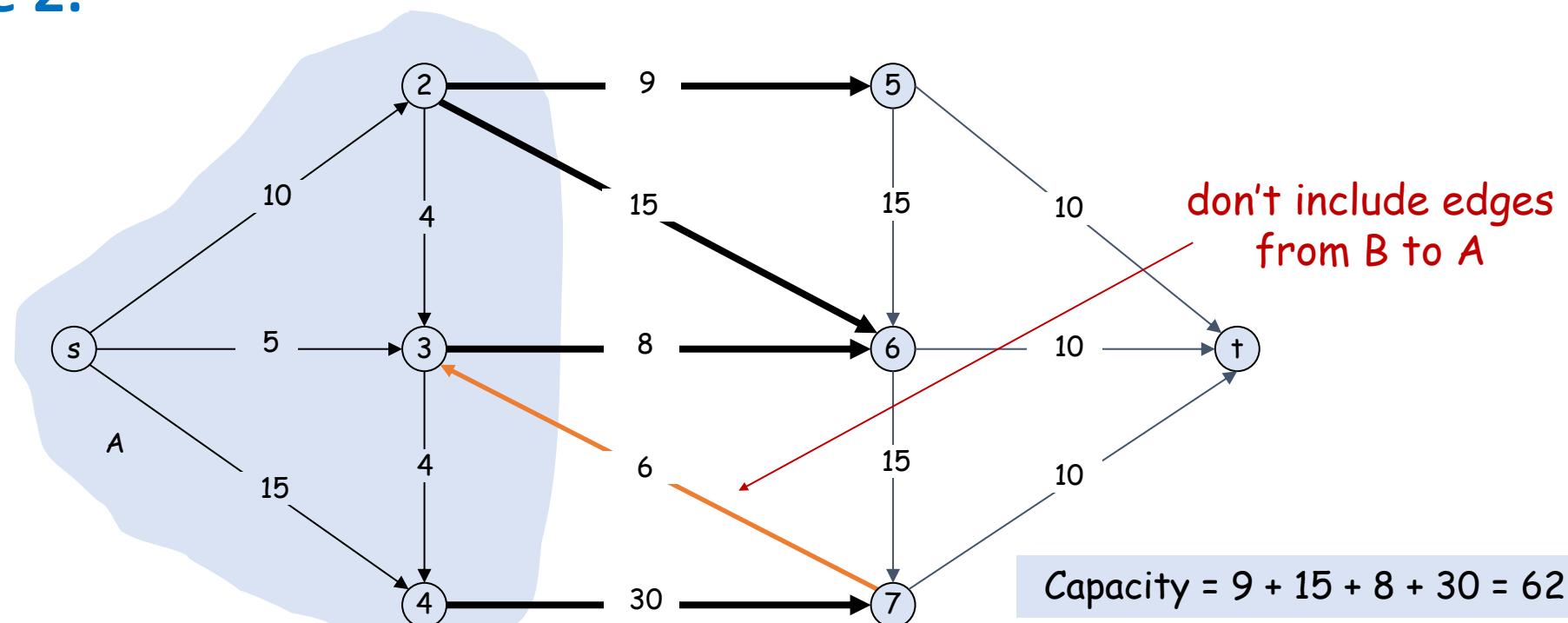
Minimum-Cut Problem

- **Def.** An st -cut (or cut) is a partition (A, B) of V with $s \in A$ and $t \in B$.
- **Def.** The capacity of a cut (A, B) is $c(A, B) = \sum_{e \text{ out of } A} c_e$
- **Example 1:**



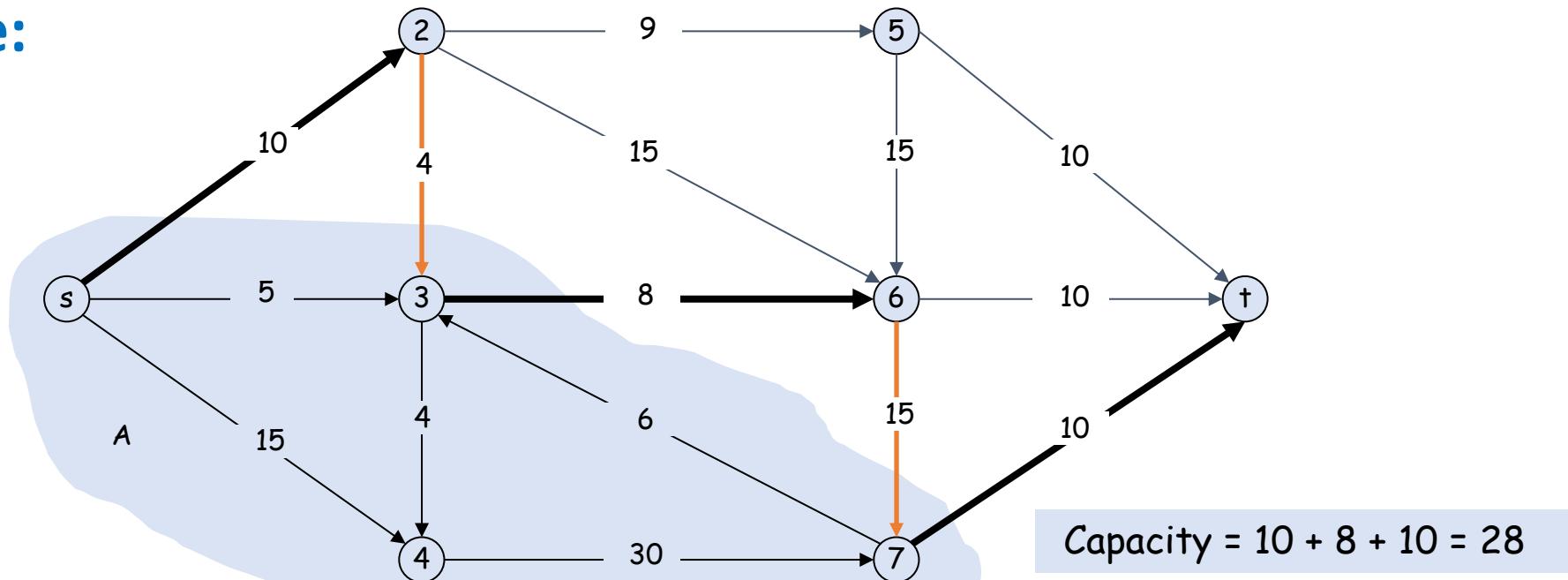
Minimum-Cut Problem

- **Def.** An st -cut (or cut) is a partition (A, B) of V with $s \in A$ and $t \in B$.
- **Def.** The capacity of a cut (A, B) is $c(A, B) = \sum_{e \text{ out of } A} c_e$
- **Example 2:**



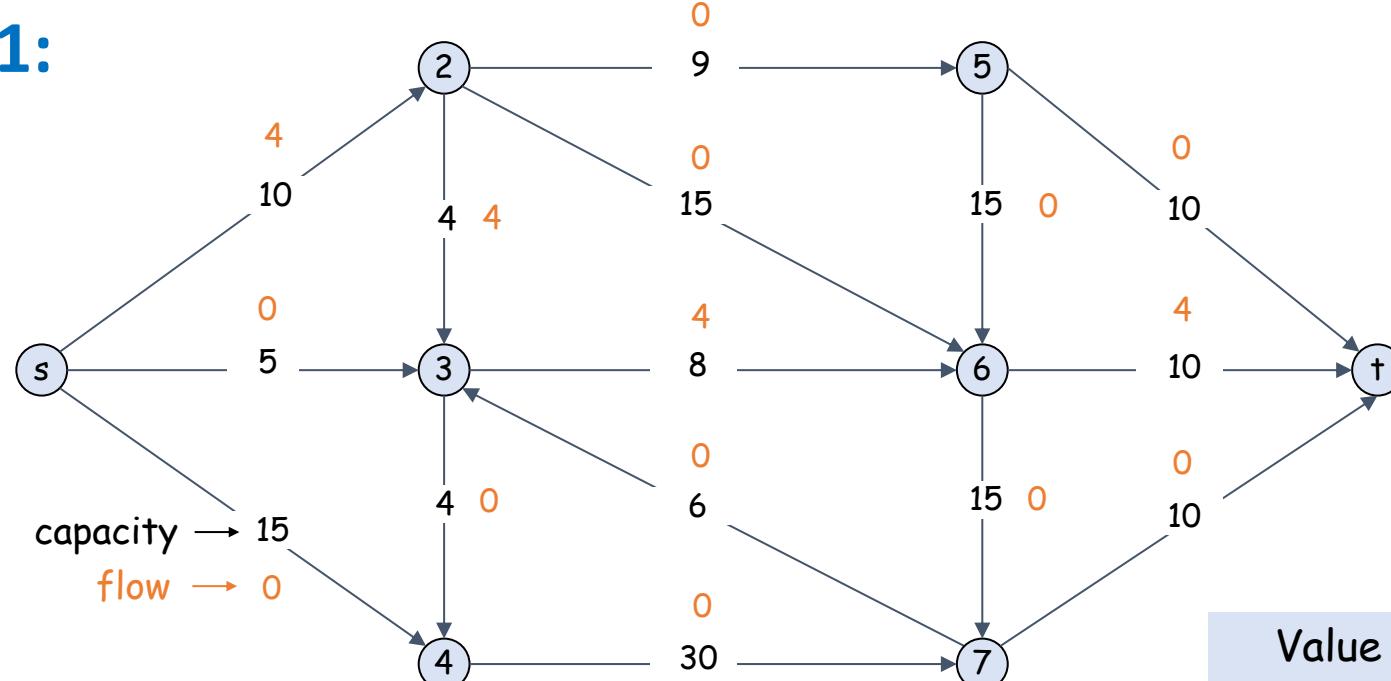
Minimum-Cut Problem

- **Def.** An st -cut (or cut) is a partition (A, B) of V with $s \in A$ and $t \in B$.
- **Def.** The capacity of a cut (A, B) is $c(A, B) = \sum_{e \text{ out of } A} c_e$
- **Min-Cut Problem.** Find a cut of minimum capacity.
- **Example:**



Maximum-Flow Problem

- **Def.** An *st-flow* (or flow) f is a function that satisfies
 - [Capacity] For each $e \in E$: $0 \leq f(e) \leq c_e$
 - [Flow conservation] For each $v \in V - \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$
- **Def.** The *value* of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e) = f^{\text{out}}(s)$.
- **Example 1:**

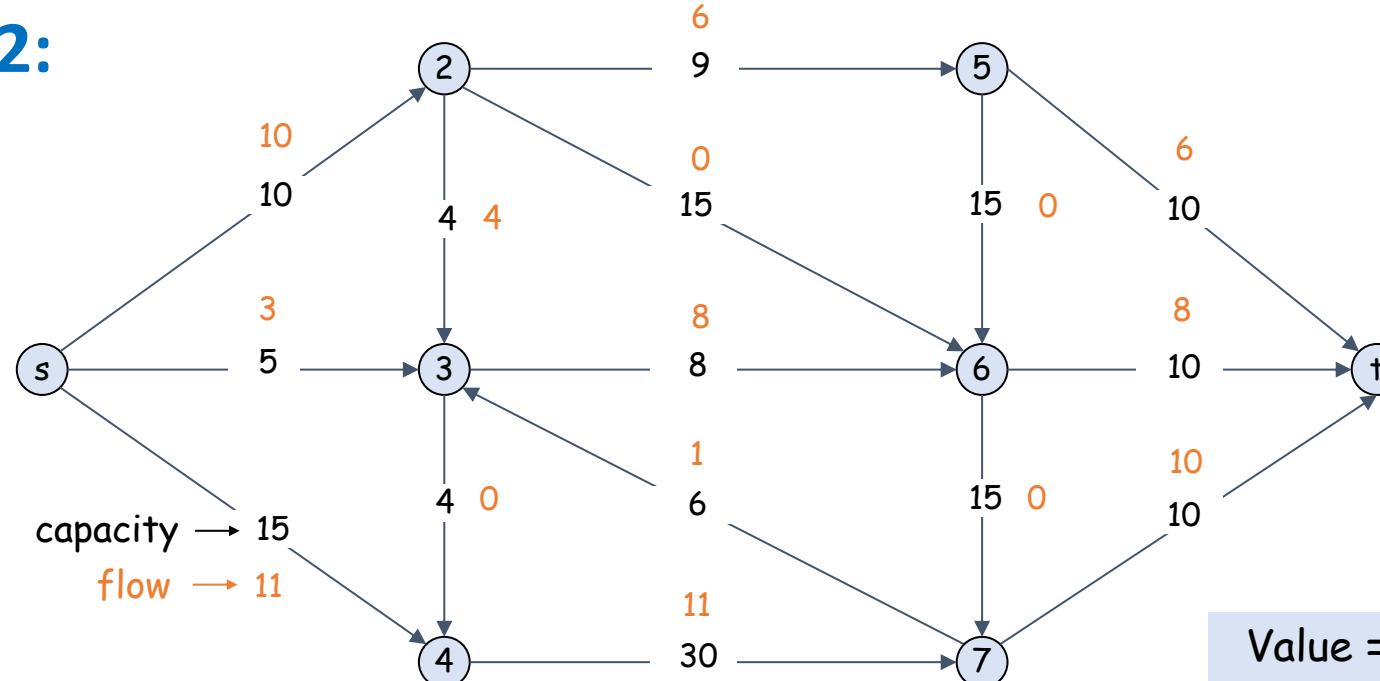


simple notation:
 $f^{\text{in}}(v) = f^{\text{out}}(v)$

↙
 进入和输出一样 (源点和终点除外)

Maximum-Flow Problem

- **Def.** An *st-flow* (or flow) f is a function that satisfies
 - [Capacity] For each $e \in E$: $0 \leq f(e) \leq c_e$
 - [Flow conservation] For each $v \in V - \{s, t\}$: $\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$
- **Def.** The *value* of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e) = f^{\text{out}}(s)$.
- **Example 2:**



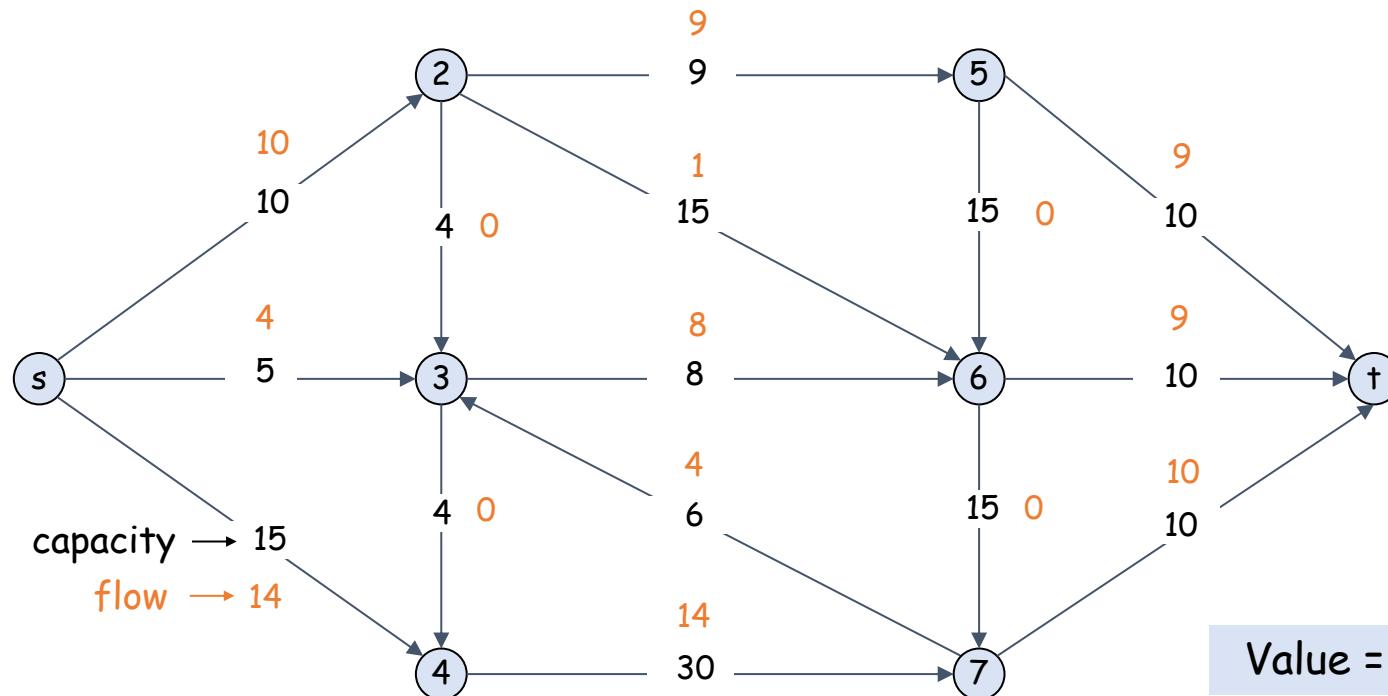
simple notation:
 $f^{\text{in}}(v) = f^{\text{out}}(v)$

$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$

Maximum-Flow Problem

- **Def.** An *st-flow* (or *flow*) f is a function that satisfies
- **Def.** The *value* of a flow f is $v(f) = \sum_{e \text{ out of } s} f(e)$
- **Max-Flow Problem.** Find a flow of *maximum value*.

- **Example:**





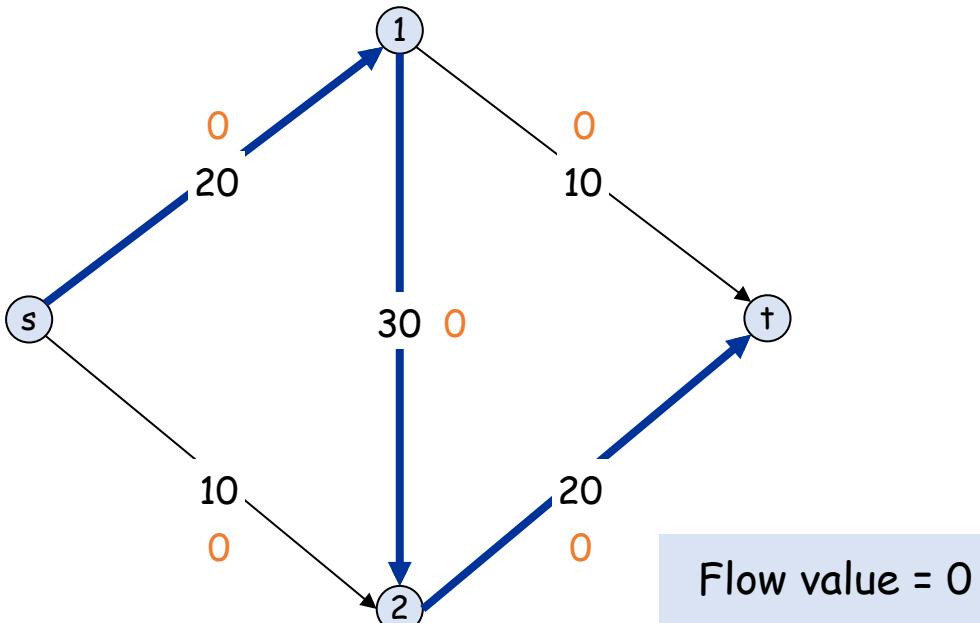
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

2. Ford-Fulkerson Algorithm

Toward a Max-Flow Algorithm

- **Greedy algorithm:**

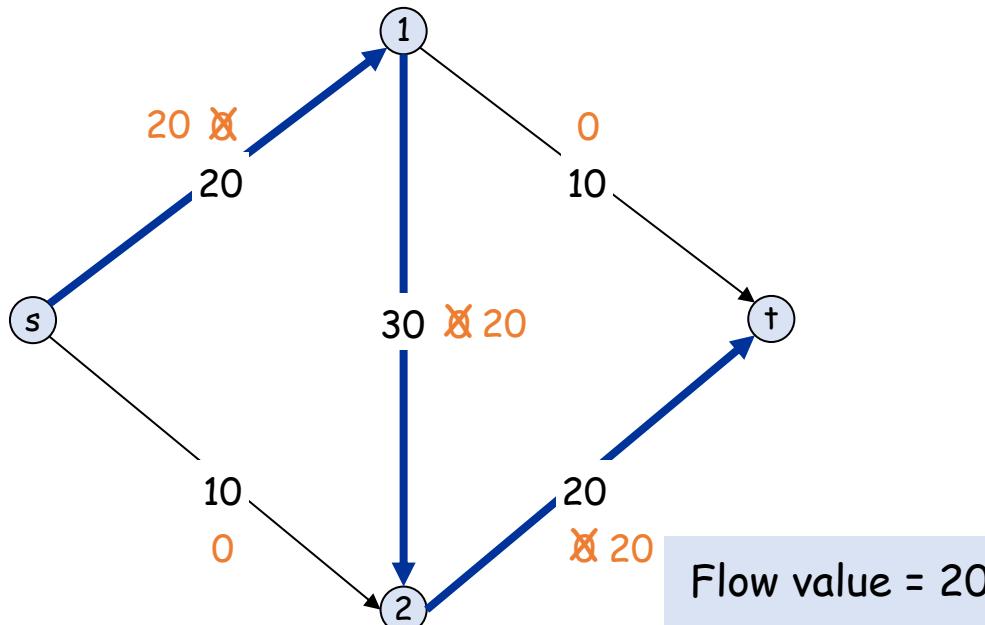
- Start with $f(e) = 0$ for all edges $e \in E$.
- Find an $s-t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get stuck.



Toward a Max-Flow Algorithm

- **Greedy algorithm:**

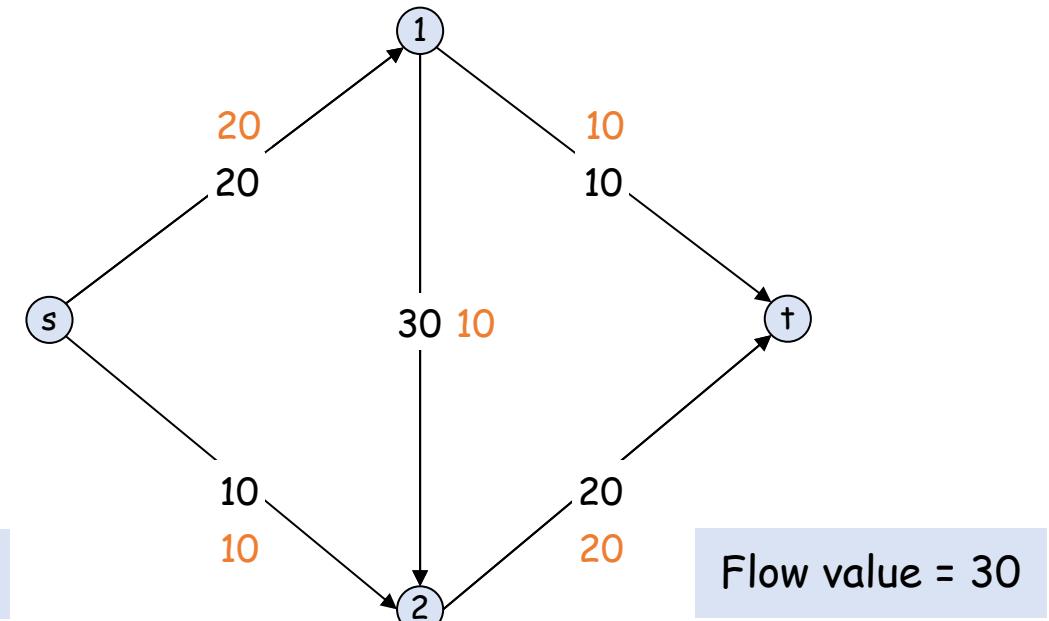
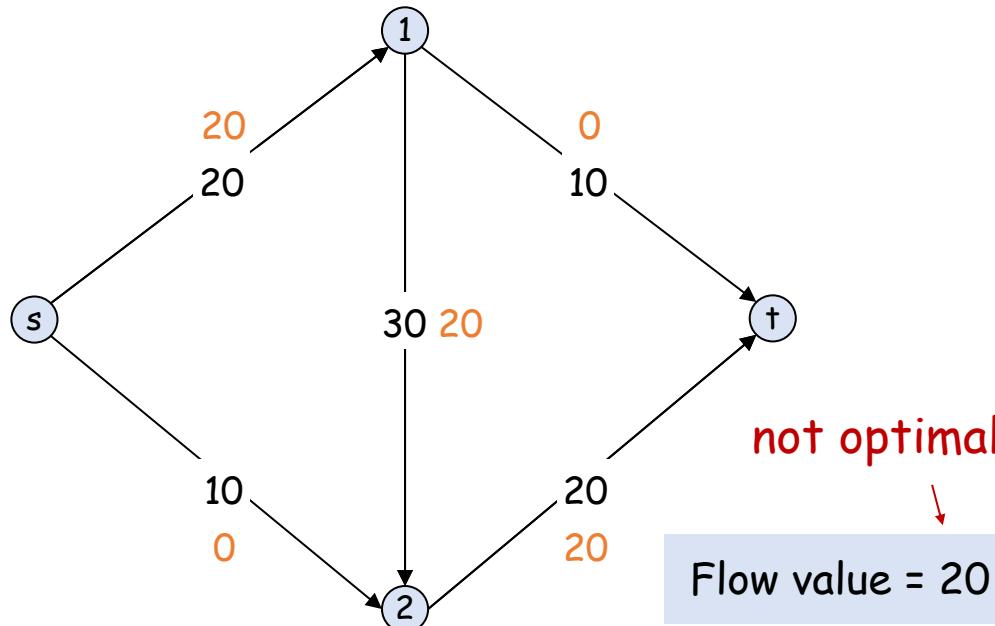
- Start with $f(e) = 0$ for all edges $e \in E$.
- Find an $s-t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get stuck.



Toward a Max-Flow Algorithm

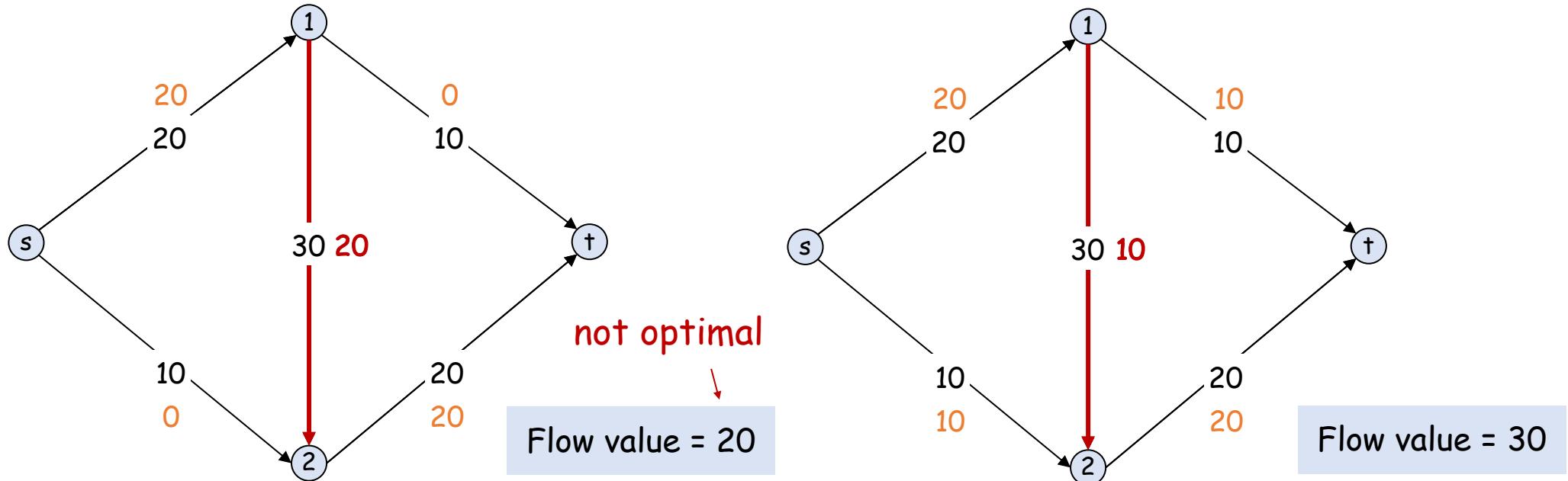
- **Greedy algorithm:**

- Start with $f(e) = 0$ for all edges $e \in E$.
- Find an $s-t$ path P where each edge has $f(e) < c(e)$.
- Augment flow along path P .
- Repeat until you get stuck.



Toward a Max-Flow Algorithm

- Q. Why does the greedy algorithm fail?
- A. Once flow on an edge is increased, it **never decreases**.
- **Bottom line.** Need some mechanism to “undo” a bad decision.





Residual Network

- **Original edge:** $e = (u, v) \in E$

➤ flow $f(e)$, capacity $c(e)$

- **Reverse edge:** $e^R = (v, u)$

➤ used to “undo” flow

- **Residual capacity:** c_f

➤ original edge: $c_f(e) = c(e) - f(e)$

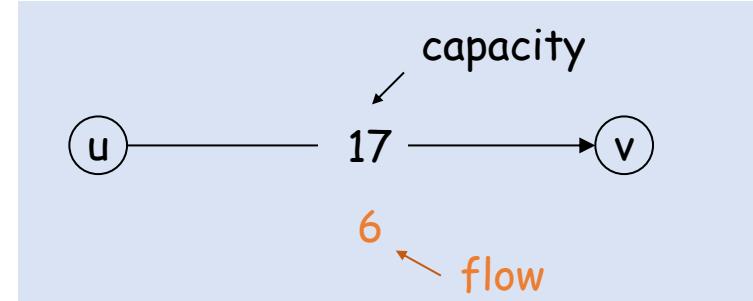
➤ reverse edge: $c_f(e^R) = f(e)$

- **Residual network:** $G_f = (V, E_f, s, t, c_f)$

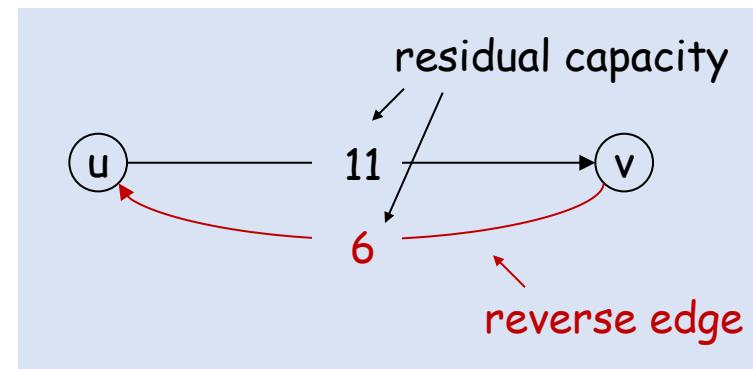
➤ $E_f = \{e: f(e) < c(e)\} \cup \{e^R: f(e) > 0\}$: edges with positive residual capacity

- **Key property.** Δf is a flow in G_f if and only if $f + \Delta f$ is a flow in G .

flow on a reverse edge negates flow on corresponding original forward edge



残余网络是基于原图构建的一个网络，它包含了所有可用的正残余容量（即流量仍然可以通过的边）和负残余容量（即反向边可以撤销的流量）





Augmenting Path

- **Def.** An augmenting path P is a simple $s-t$ path in the residual network G_f .
- **Def.** The bottleneck capacity of an augmenting path P in G_f , denoted by $\text{bottleneck}(G_f, P)$, is the minimum residual capacity of edges in P .
- **Key property.** Consider a flow network $G = (V, E, s, t, c)$ with flow f and augmenting path P in G_f . After calling the following $\text{Augment}(f, c, P)$, the returned f' is a flow in G and $v(f') = v(f) + \text{bottleneck}(G_f, P)$.

```
Augment(f, c, P) {
    δ = bottleneck(Gf, P)
    foreach e ∈ P {
        if (e ∈ E) f(e) = f(e) + δ
        else          f(eR) = f(eR) - δ
    }
    return f
}
```

↑
e is a reverse edge in G_f
so e^R is an edge in G



Ford-Fulkerson Algorithm

- **Ford-Fulkerson (FF) algorithm.**



- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s-t$ path P in the residual network G_f .
- Augment flow along path P .
- Repeat until you get stuck.

```
Ford-Fulkerson(G, s, t, c) {
    foreach e ∈ E: f(e) = 0
    Gf = residual network of G with respect to flow f

    while (there exists an augmenting path P in Gf) {
        f = Augment(f, c, P)
        update Gf
    }
    return f
}
```



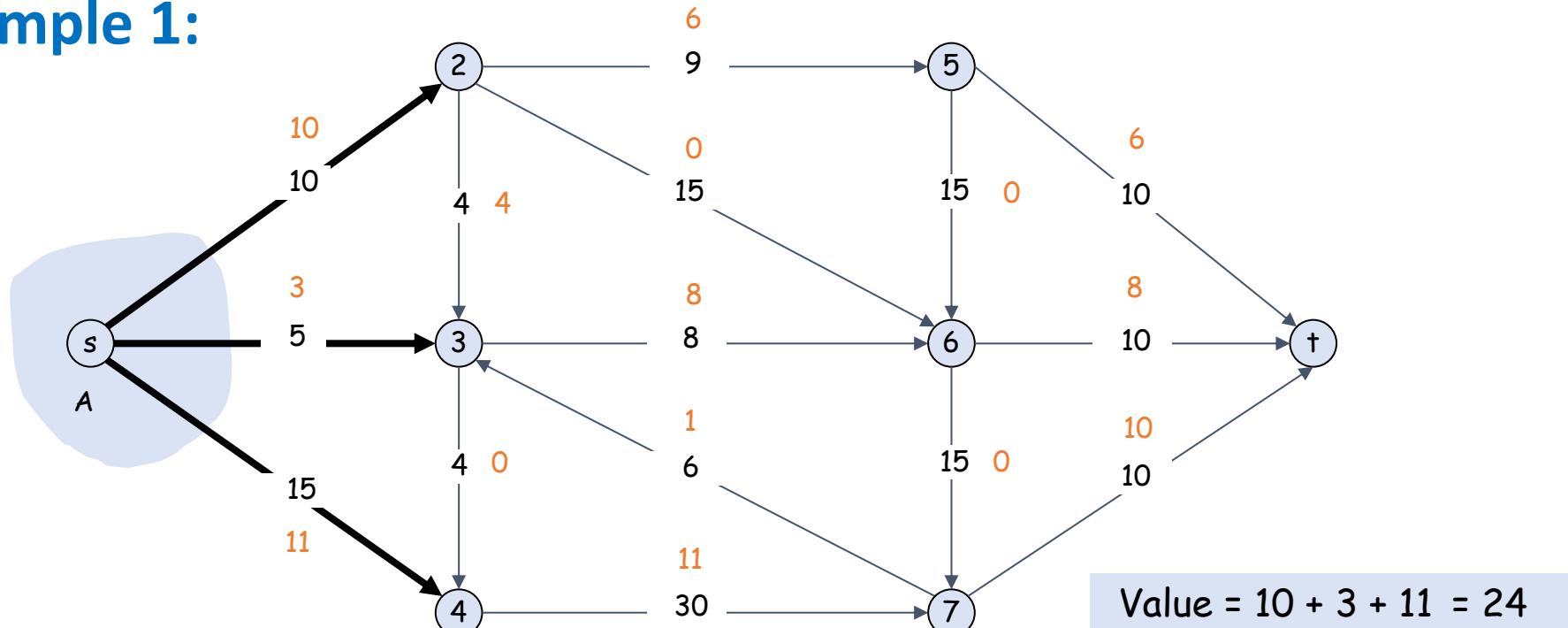
3. Max-Flow Min-Cut Theorem

Relationship between Flows and Cuts

- **Flow Value Lemma.** Let f be any flow, and let (A, B) be any cut. Then, the value of the flow f equals the **net flow** across the cut (A, B) .

$$v(f) = f^{out}(A) - f^{in}(A)$$

- **Example 1:**

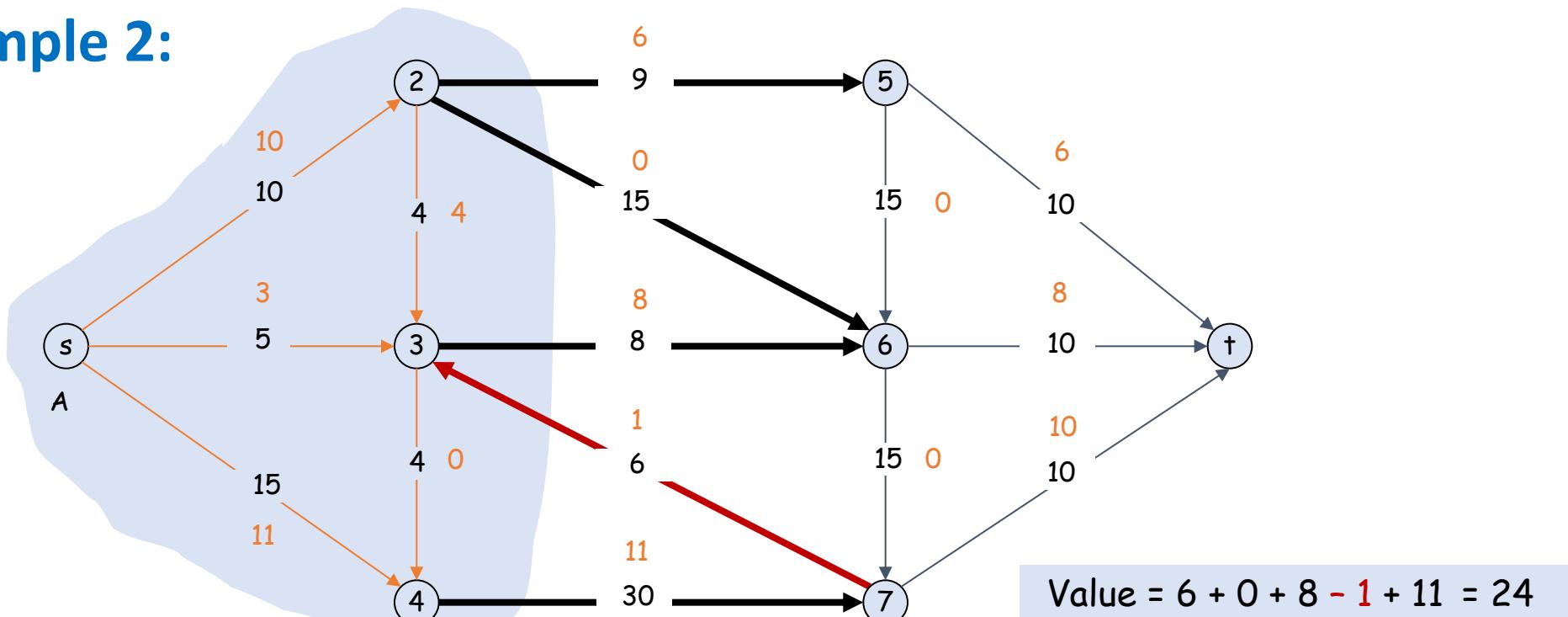


Relationship between Flows and Cuts

- **Flow Value Lemma.** Let f be any flow, and let (A, B) be any cut. Then, the value of the flow f equals the **net flow** across the cut (A, B) .

$$v(f) = f^{out}(A) - f^{in}(A)$$

- **Example 2:**

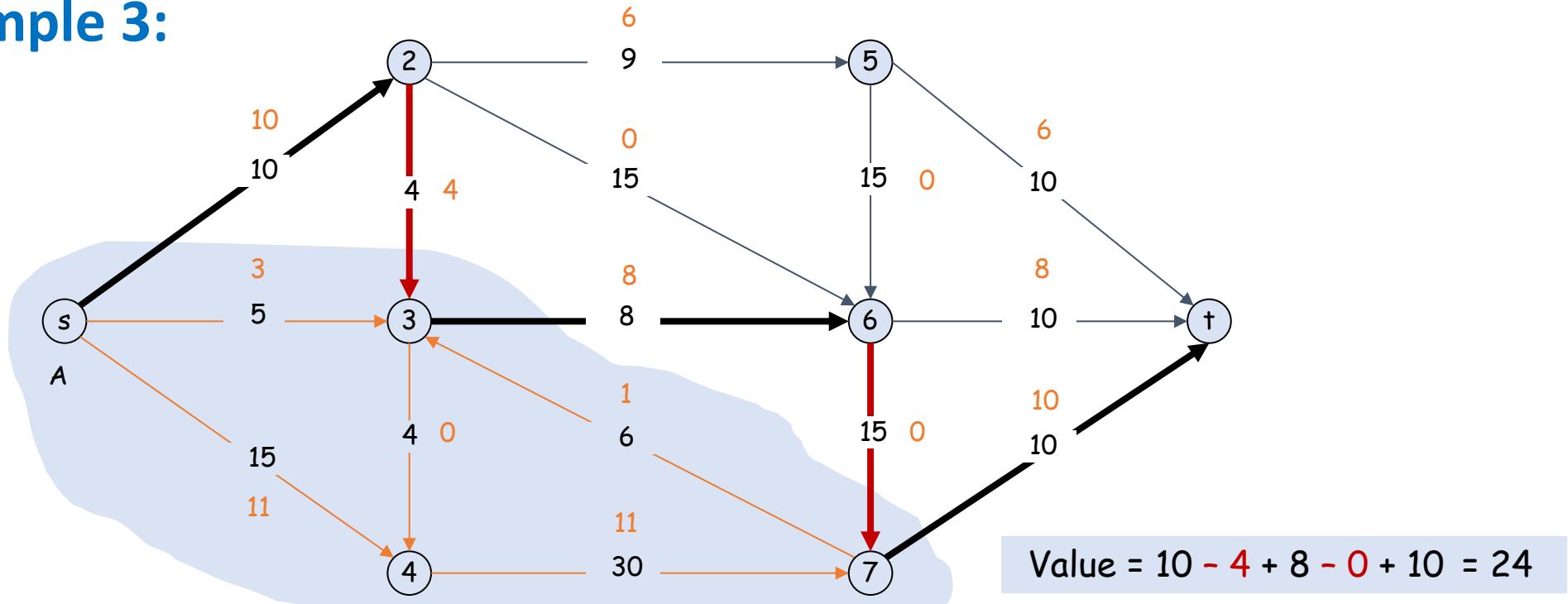


Relationship between Flows and Cuts

- **Flow Value Lemma.** Let f be any flow, and let (A, B) be any cut. Then, the value of the flow f equals the **net flow** across the cut (A, B) .

$$v(f) = f^{out}(A) - f^{in}(A)$$

- **Example 3:**





Relationship between Flows and Cuts

- **Flow Value Lemma.** Let f be any flow, and let (A, B) be any cut. Then, the value of the flow f equals the **net flow** across the cut (A, B) .

$$v(f) = f^{out}(A) - f^{in}(A) \quad v(f) = f^{in}(B) - f^{out}(B)$$

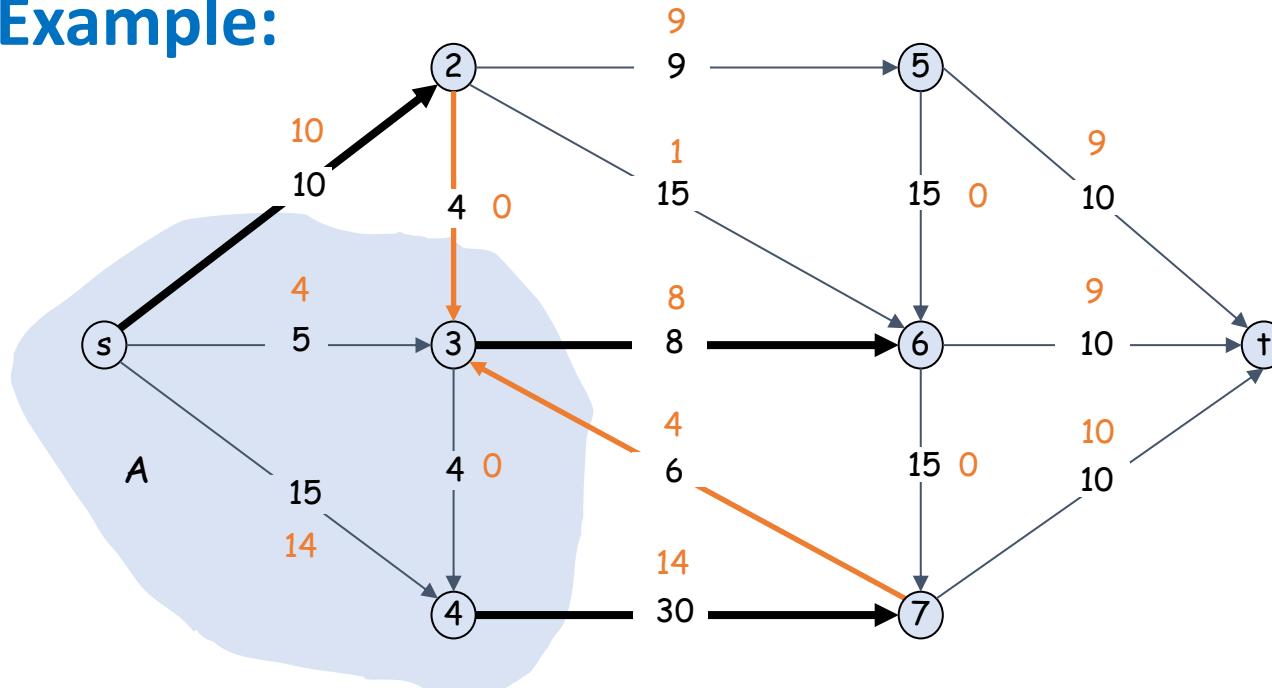
- **Pf.**

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } s} f(e) = f^{out}(s) = f^{out}(s) - f^{in}(s) \xrightarrow{\text{f}^{in}(s) = 0} \\ &= \sum_{v \in A} (f^{out}(v) - f^{in}(v)) \xrightarrow{\substack{\text{by flow conservation, } f^{out}(v) - f^{in}(v) = 0 \\ \text{for all } v \text{ in } A \text{ except for } v = s}} \\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{out}(A) - f^{in}(A) \end{aligned}$$

Relationship between Flows and Cuts

- **Weak Duality.** Let f be any flow, and let (A, B) be any cut. Then the value of the flow f is at most the capacity of the cut: $v(f) \leq c(A, B)$.
 - Recall that $c(A, B) = \sum_{e \text{ out of } A} c_e$.

- **Example:**



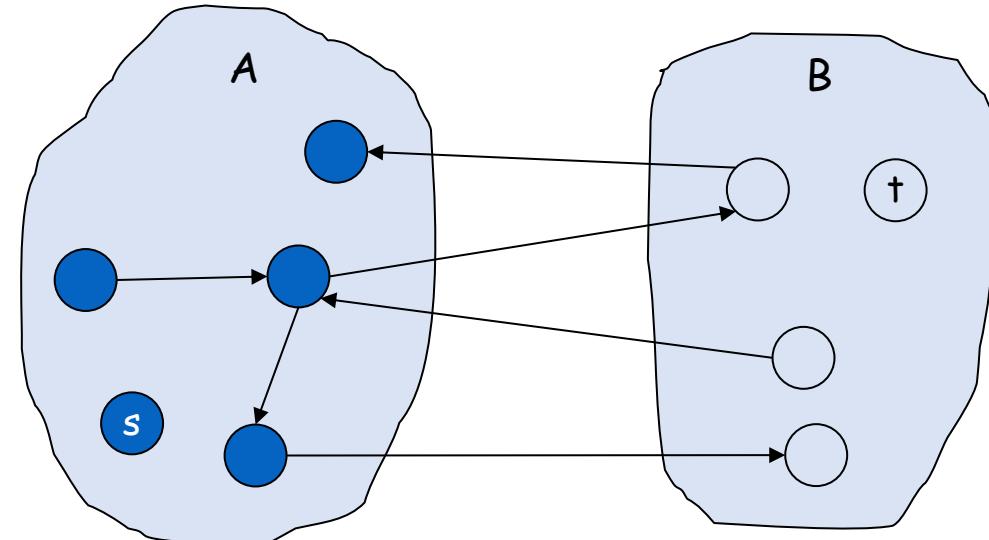
Flow value = 28
 Cut capacity = 48 \Rightarrow Flow value ≤ 48

Relationship between Flows and Cuts

- **Weak Duality.** Let f be any flow, and let (A, B) be any cut. Then the value of the flow f is at most the capacity of the cut: $v(f) \leq c(A, B)$.
 - Recall that $c(A, B) = \sum_{e \text{ out of } A} c_e$.

- **Pf.** $v(f) = f^{out}(A) - f^{in}(A)$

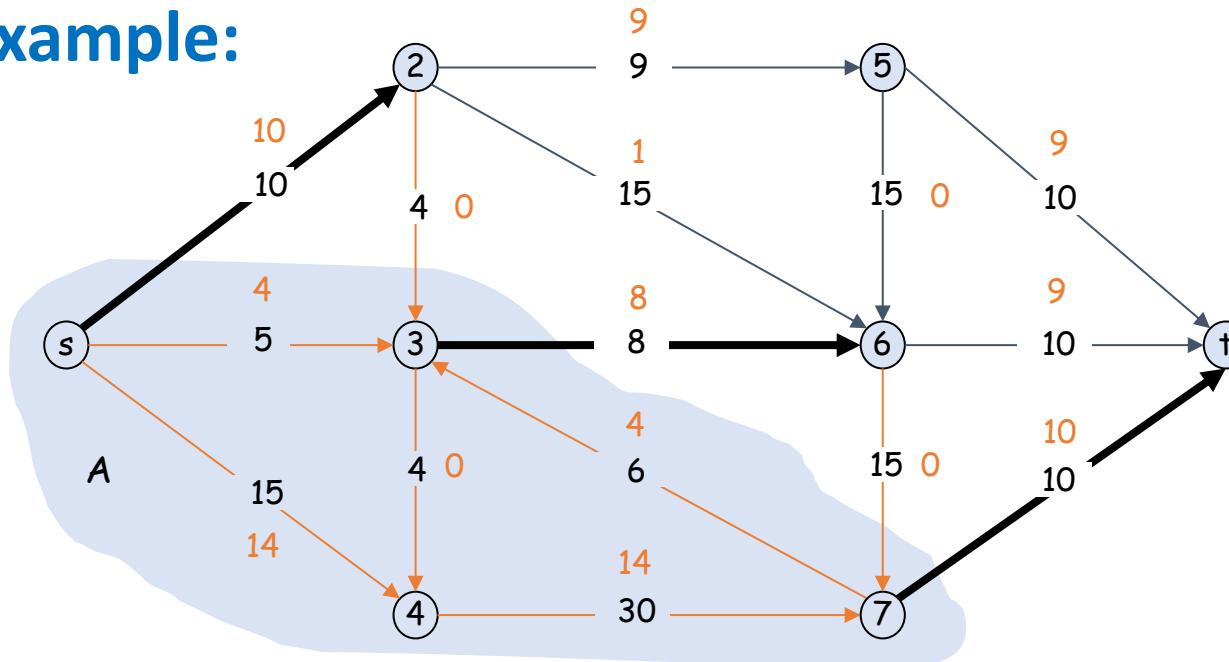
$$\begin{aligned} &\leq f^{out}(A) \\ &\stackrel{\substack{\text{flow value} \\ \text{lemma}}}{=} \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= c(A, B) \end{aligned}$$



Certificate of Optimality

- **Corollary.** Let f be any flow, and let (A, B) be any cut. If $v(f) = c(A, B)$, then f is a **max flow** and (A, B) is a **min cut**.
- **Pf.** Every flow value is **upper bounded** by every cut capacity!

- **Example:**



Flow value = 28
Cut capacity = 28 \Rightarrow Flow value ≤ 28



Max-Flow Min-Cut Theorem

- **Max-Flow Min-Cut Theorem.** [Ford-Fulkerson 1956] Value of a max flow is **equal** to capacity of a min cut.
- **Augmenting Path Theorem.** Flow f is a **max flow** iff no augmenting paths.
- **Pf.** We prove both by showing the following are equivalent:
 - (i) There exists a cut (A, B) such that $v(f) = c(A, B)$.
 - (ii) f is a max flow.
 - (iii) There is no augmenting path with respect to f .

(i) \Rightarrow (ii): This was the previous **Corollary**.

(ii) \Rightarrow (iii): We prove the **contrapositive**.
➤ Let f be a flow. If there exists an augmenting path, then we can improve flow f by sending flow along this path. Then, f is not a max flow.



Max-Flow Min-Cut Theorem

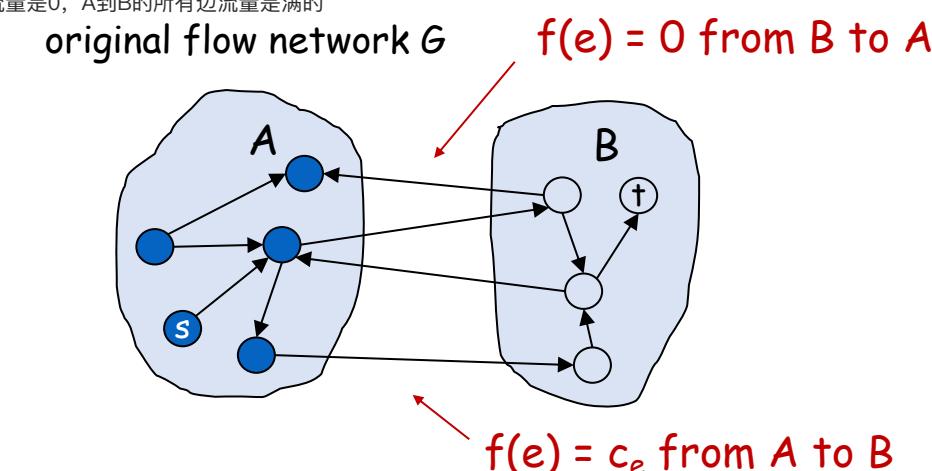
- Pf. (continued)

- (i) There exists a cut (A, B) such that $v(f) = c(A, B)$.
- (iii) There is no augmenting path with respect to f .

(iii) \Rightarrow (i):

- Let f be a flow with no augmenting paths.
 - Let A be the set of nodes **reachable from s** in residual network G_f .
 - By definition of $A \Rightarrow s \in A$. No augmenting path for $f \Rightarrow t \notin A$.
- given a max flow f (so no augmenting path)
can find a min cut (A, B) in $O(m)$ time
肯定从A不能直接到B, 否则s能到t, 也就是有增广路

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ \text{flow value lemma} &= \sum_{e \text{ out of } A} c(e) - 0 \\ &= c(A, B) \end{aligned}$$





4. Capacity-Scaling Algorithm



Ford-Fulkerson Algorithm: Analysis

不是整数，可能导致不断逼近极限并且不是最优值

- **Assumption.** Every edge capacity c_e is an **integer** between **1** and **C**.
- **Integrality Invariant.** Throughout FF, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.
- **Theorem.** FF terminates after at most $v(f^*) \leq nC$ augmenting paths, where f^* is a **max flow**. (Assume flow network has no parallel edges.)
- **Pf.** Consider cut where $A = \{s\}$ and note that each augmentation increases the value of the flow by at least **1**. ▀
- **Corollary.** The running time of Ford-Fulkerson is $O(mnC)$.
- **Pf.** Can use either **BFS** or **DFS** to find an augmenting path in $O(m)$ time. ▀

最多是遍历所有顶点



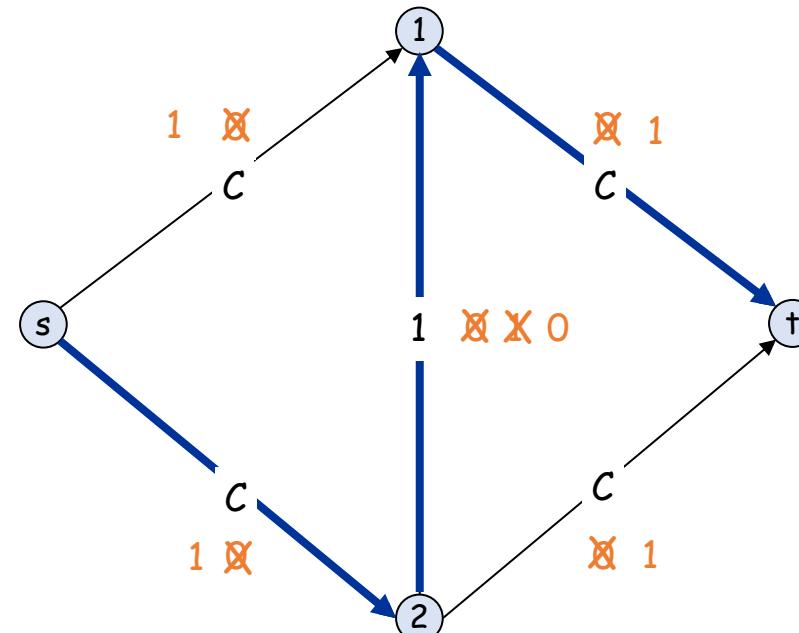
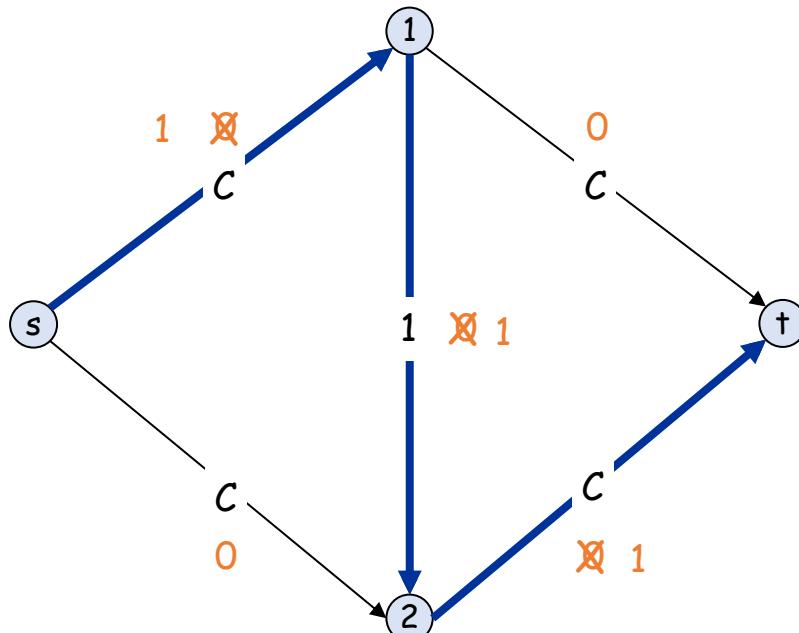
Ford-Fulkerson Algorithm: Analysis

- **Assumption.** Every edge capacity c_e is an **integer** between **1** and **C** .
- **Integrality Invariant.** Throughout **FF**, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.
- **Theorem.** **FF** terminates after at most $v(f^*) \leq nC$ augmenting paths, where f^* is a **max flow**. (Assume flow network has no parallel edges.)
- **Corollary.** The running time of **Ford-Fulkerson** is $O(mnC)$.
- **Integrality Theorem.** There exists an **integral max flow f^*** .
每条边的流量都是整数
- **Pf.** Since **FF** always terminates if capacities are integral, theorem follows from **integrality invariant** (and **augmenting path theorem**). ▀

Ford-Fulkerson: Exponential Example

- Q. Is Ford-Fulkerson algorithm polynomial in input size? $\leftarrow m, n, \log_2 C$
- A. No. If max capacity is C , then algorithm can take $2C$ iterations.

$$C = 2^{\{\log_2 C\}}$$





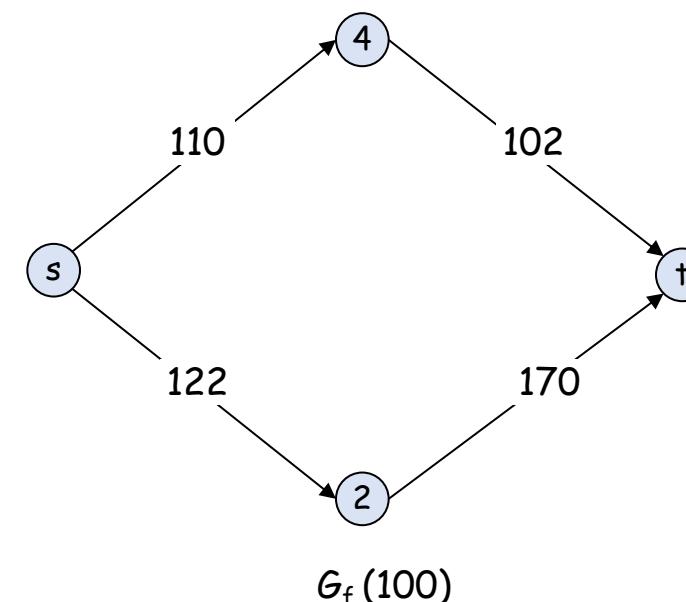
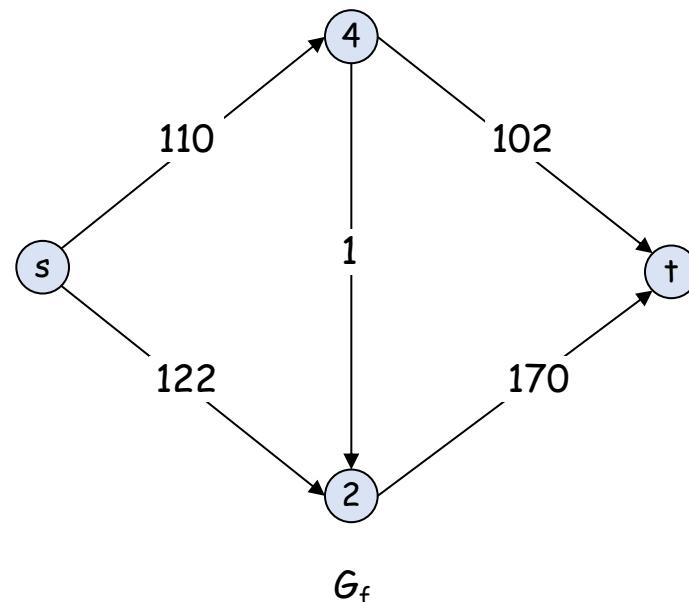
Choosing Good Augmenting Paths

- **Caveat.** If capacities can be **irrational**, FF may **not terminate** or converge!
- **Use care when selecting augmenting paths:**
 - Some choices lead to **exponential** algorithms.
 - Clever choices lead to **polynomial** algorithms.
- **Goal.** Choose augmenting paths so that:
 - Can find augmenting paths efficiently.
 - Few iterations.
- **Choose augmenting paths with:**
 - **Max** bottleneck capacity (“fattest”). ← **difficult, how to find?**
 - **Sufficiently large** bottleneck capacity. ← **coming next**
 - **Fewest edges.** [Edmonds-Karp 1972, Dinitz 1970] ← **later sections**

capacities are rational in practice
but FF could run in exponential time

Capacity-Scaling Algorithm

- **Overview.** Choosing augmenting paths with “large” bottleneck capacity.
 - Maintain scaling parameter Δ .
 - Let $G_f(\Delta)$ be the subnetwork of the residual network containing only those edges with capacity $\geq \Delta$.
 - Any augmenting path in $G_f(\Delta)$ has bottleneck capacity $\geq \Delta$.
not necessarily largest





Capacity-Scaling Algorithm

- **Greedy algorithm.** Halve Δ when no augmenting path can be found.

```
Capacity-Scaling(G, s, t, c) {
    foreach e ∈ E: f(e) = 0
    Δ = largest power of 2 that is ≤ c
    Gf = residual network with respect to flow f

    while (Δ ≥ 1) {
        Gf(Δ) = Δ-residual network of G with respect to flow f
        while (there exists an augmenting path P in Gf(Δ)) {
            f = Augment(f, c, P)
            update Gf(Δ)
        }
        Δ = Δ / 2
    }
    return f
}
```



Capacity-Scaling Algorithm: Correctness

- **Assumption.** Every edge capacity c_e is an integer between 1 and C .
- **Invariant.** The scaling parameter Δ is a power of 2.
- **Integrality Invariant.** Throughout capacity-scaling algorithm, every edge flow $f(e)$ and residual capacity $c_f(e)$ are integers.

到最后 $\Delta=1$, 那么就和原图的剩余网络一样

- **Theorem.** If capacity-scaling algorithm terminates, then f is a max flow.
- **Pf. (direct proof)**
 - **Integrality Invariant** $\Rightarrow G_f(1) = G_f$.
 - Upon termination of the $\Delta = 1$ phase, there are no augmenting paths.
 - **Augmenting Path Theorem** \Rightarrow the resulting flow f is a max flow. ■



Capacity-Scaling Algorithm: Running Time

- **Lemma 1.** The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.
- **Pf.** Initially $C/2 < \Delta \leq C$; Δ decreases by a factor of 2 each iteration. ▀
- **Lemma 2.** Let f be the flow at the end of a Δ -scaling phase. Then the value of a maximum flow $\leq v(f) + m\Delta$. (Proof shown later.)
- **Pf idea.** Adding edges with capacity $< \Delta$ can only increase flow by $\leq m\Delta$.
- **Lemma 3.** There are $\leq 2m$ augmentations per scaling phase.
- **Pf.** Let f be the flow at the end of the previous scaling phase $\Delta' = 2\Delta$.
 - **Lemma 2** \Rightarrow maximum flow value $\leq v(f) + m\Delta' = v(f) + m(2\Delta)$.
 - Each augmentation in a Δ -scaling phase increases $v(f)$ by at least Δ .
For the first Δ -scaling phase, since by selection of Δ we have $C < 2\Delta$, the maximum flow value is at most $mC \leq m(2\Delta)$. Same proof holds. ▀



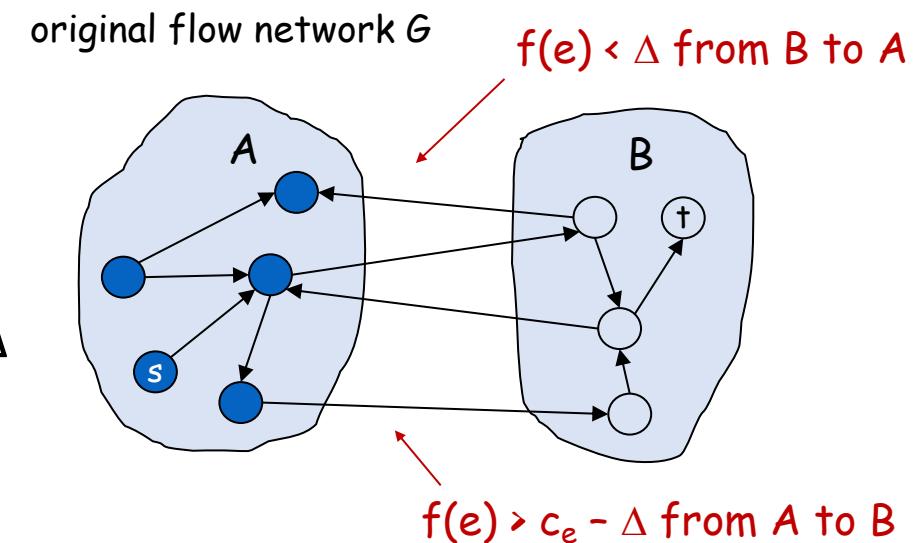
Capacity-Scaling Algorithm: Running Time

- **Lemma 1.** The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.
- **Lemma 3.** There are $\leq 2m$ augmentations per scaling phase.
- **Theorem.** The capacity-scaling algorithm takes $O(m^2 \log C)$ time.
- **Pf.** **Lemma 1 + Lemma 3 $\Rightarrow O(m \log C)$** augmentations. Finding an augmenting path takes $O(m)$ time. ▀

Capacity-Scaling Algorithm: Running Time

- **Lemma 2.** Let f be the flow at the end of a Δ -scaling phase. Then the value of a maximum flow $\leq v(f) + m\Delta$.
- **Pf.** (similar to the proof of Max-Flow Min-Cut Theorem)
 - We show that there exists a cut (A, B) such that $c(A, B) \leq v(f) + m\Delta$.
 - Choose A to be the set of nodes **reachable** from s in $G_f(\Delta)$.
 - By definition of $A \Rightarrow s \in A$. No augmenting path for $f \Rightarrow t \notin A$.

$$\begin{aligned}
 v(f) &= f^{out}(A) - f^{in}(A) \\
 \text{flow value lemma} &\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\
 &\geq \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\
 &\geq c(A, B) - m\Delta
 \end{aligned}$$





南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

5. Edmonds-Karp Algorithm



Shortest Augmenting Path (Edmonds-Karp)

- Q. How to choose next augmenting path in Ford-Fulkerson?
- A. Pick a path that uses the **fewest edges**. ← can find via BFS
- **Edmonds-Karp (EK) algorithm:**

```
Edmonds-Karp(G, s, t, c) {
    foreach e ∈ E: f(e) = 0
    Gf = residual network of G with respect to flow f

    while (there exists an augmenting path P in Gf) {
        P = Breath-First-Search(Gf)
        f = Augment(f, c, P)
        update Gf
    }
    return f
}
```

core procedure



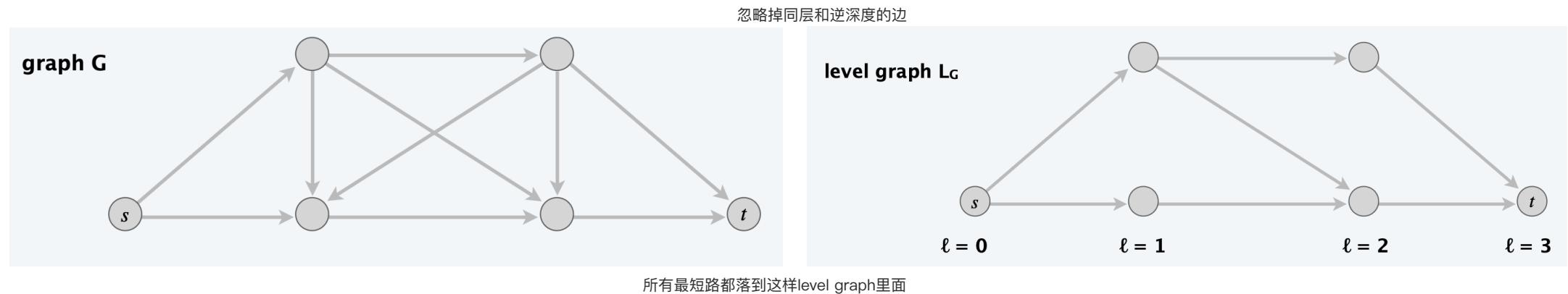
Edmonds-Karp Algorithm: Analysis Overview

- **Lemma 1.** The length of a shortest augmenting path **never decreases.** (Proof shown later.)
- **Lemma 2.** After at most m shortest-path augmentations, the length of a shortest augmenting path **strictly increases.** (Proof shown later.)
- **Theorem.** Edmonds-Karp algorithm takes $O(m^2n)$ time.
- **Pf.** (direct proof)
 - $O(m)$ time to find a shortest augmenting path via **BFS**.
 - There are $\leq mn$ augmentations.
 - ✓ Augmenting paths are **simple** \Rightarrow at most $n - 1$ different lengths
 - ✓ **Lemma 1 + Lemma 2** \Rightarrow at most m augmenting paths for each length ▀



Edmonds-Karp Algorithm: Analysis

- **Def.** Given a directed graph $G = (V, E)$ with source s , its **level graph** is defined by:
 - $\ell(v) = \text{number of edges in shortest } s\text{-}v \text{ path.}$
 - $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v, w) \in E$ such that $\ell(w) = \ell(v) + 1$.



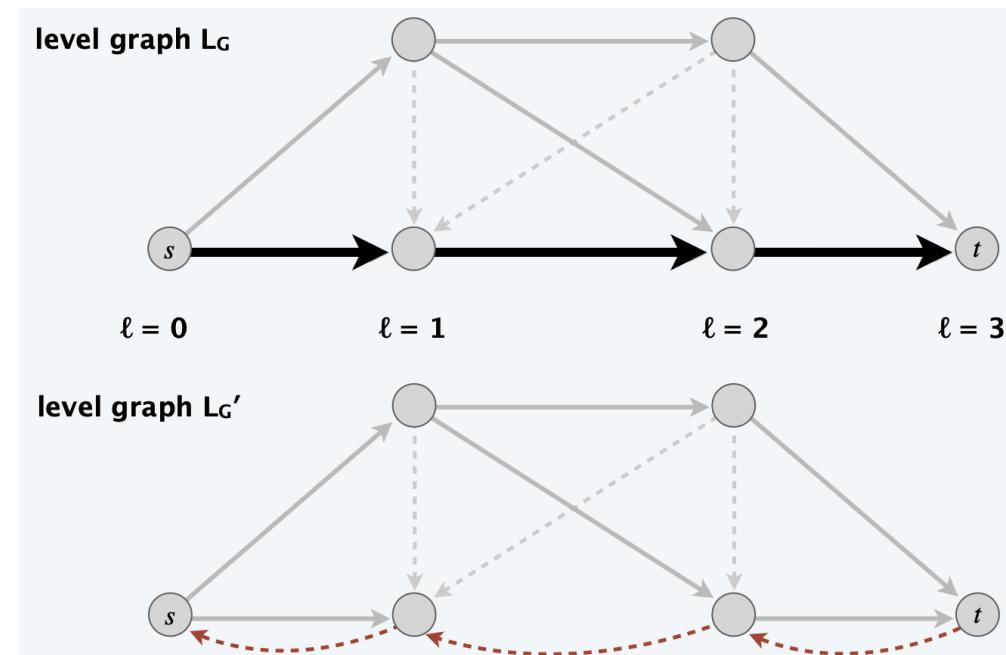
- **Key property.** P is a **shortest s - v path** in G iff P is an s - v path in L_G .

↗
all possible shortest s - v paths are captured in L_G



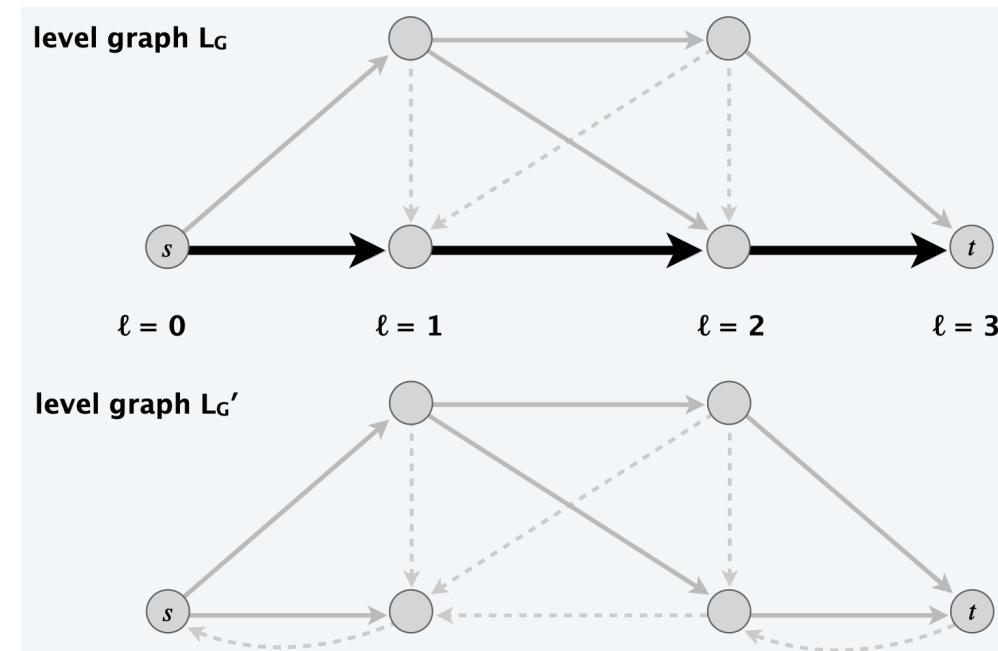
Edmonds-Karp Algorithm: Analysis

- **Pf of Lemma 1:** (Length of a shortest augmenting path never decreases.)
 - Let f and f' be flows before and after a shortest-path augmentation.
 - Let L_G and $L_{G'}$ be level graphs of G_f and $G_{f'}$. Only reverse edges added to $G_{f'}$.
 - Any $s-t$ path that uses a reverse edge is longer than previous length. ▀



Edmonds-Karp Algorithm: Analysis

- **Pf of Lemma 2:** (After at most m shortest-path augmentations, the length of a shortest augmenting path strictly increases.)
 - At least one (bottleneck) edge is deleted from L_G per augmentation.
 - No new edge added to L_G until it does not have any $s-t$ path, then shortest path length in the new level graph strictly increases. ▀





Edmonds-Karp Algorithm: Summary

- **Lemma 1.** The length of a shortest augmenting path **never decreases**.
- **Lemma 2.** After at most m shortest-path augmentations, the length of a shortest augmenting path **strictly increases**.
- **Theorem.** Edmonds-Karp algorithm takes $O(m^2n)$ time.
- **Fact.** $\Theta(mn)$ augmentations are **necessary** for some flow networks.
- **Solution.** Try to **decrease time per augmentation** instead.
 - Simple idea $\Rightarrow O(mn^2)$ [Dinitz 1970] ← next section invented by Dinitz in response to a class exercise by Adel'son-Vel'skiĭ
 - Dynamic trees $\Rightarrow O(mn \log n)$ [Sleator-Tarjan 1982]



6. Dinitz's Algorithm



Dinitz's Algorithm

- **Two types of augmentations:**

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

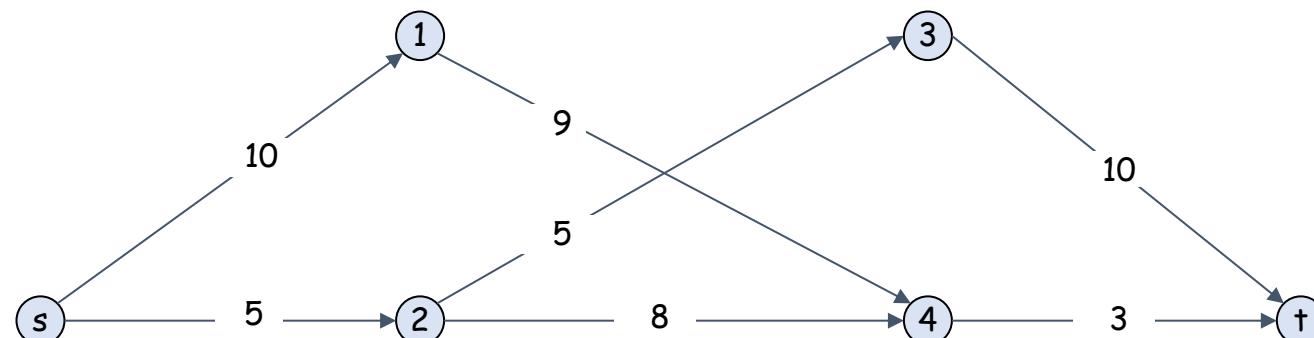
再normal状态下做剪枝

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
- If get stuck, delete node from L_G and retreat to previous node.

Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.
- **Phase of normal augmentations:**
 - Construct level graph L_G .
 - Start at s , advance along an edge in L_G until reach t or get stuck.
 - If reach t , augment flow; update L_G ; and restart from s .
 - If get stuck, delete node from L_G and retreat to previous node.

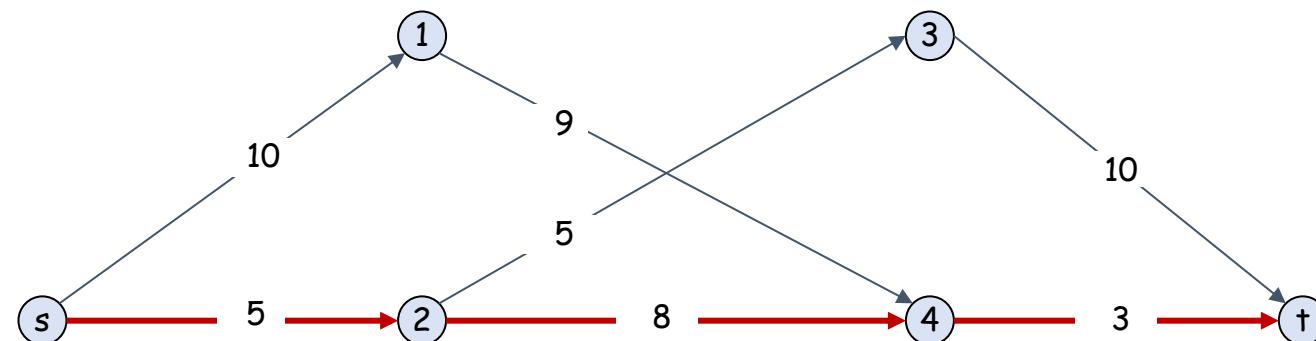


Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
- If get stuck, delete node from L_G and retreat to previous node.





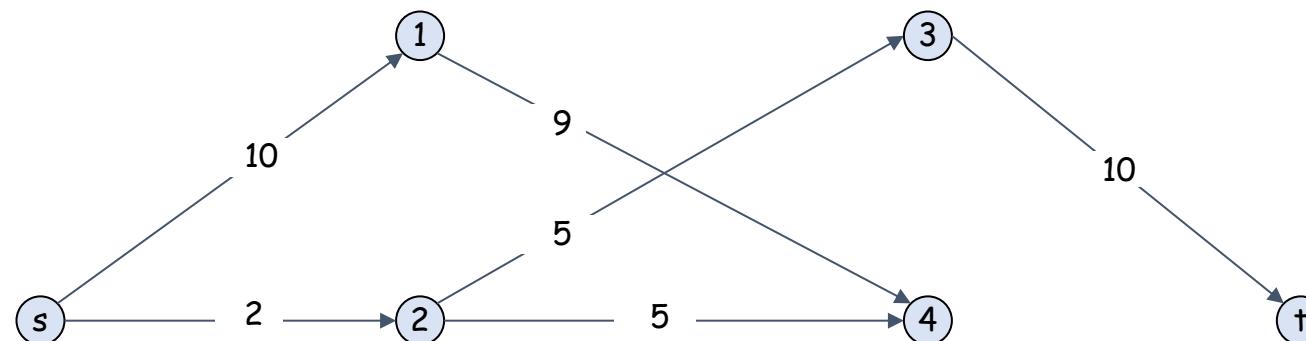
Dinitz's Algorithm

- **Two types of augmentations:**

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
增广的反向边并不能使路径更长，所以就不用加进去
- If get stuck, delete node from L_G and retreat to previous node.



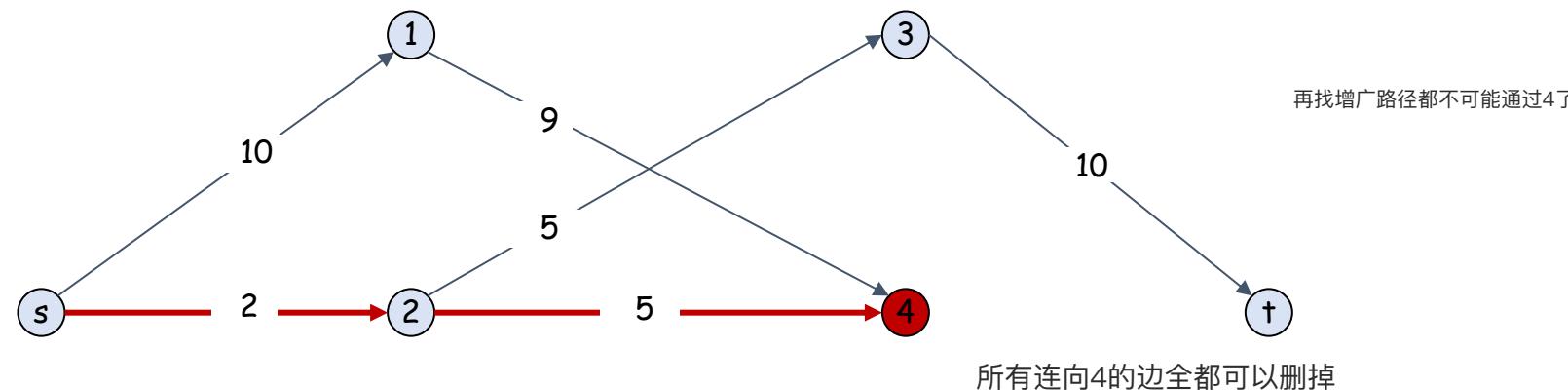


Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- Phase of normal augmentations:

- Construct level graph L_G .
 - Start at s , advance along an edge in L_G until reach t or get stuck.
 - If reach t , augment flow; update L_G ; and restart from s .
 - If get stuck, delete node from L_G and retreat to previous node.

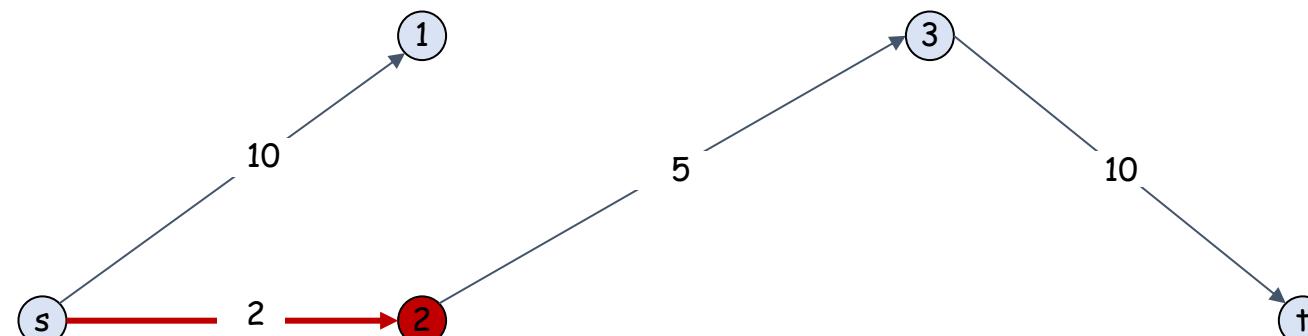




Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**
 - Construct level graph L_G .
 - Start at s , advance along an edge in L_G until reach t or get stuck.
 - If reach t , augment flow; update L_G ; and restart from s .
 - If get stuck, delete node from L_G and retreat to previous node.

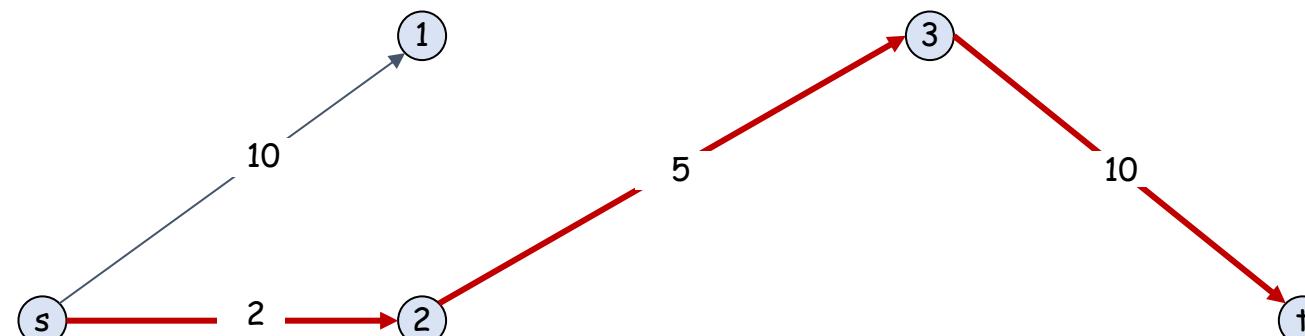


Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
- If get stuck, delete node from L_G and retreat to previous node.



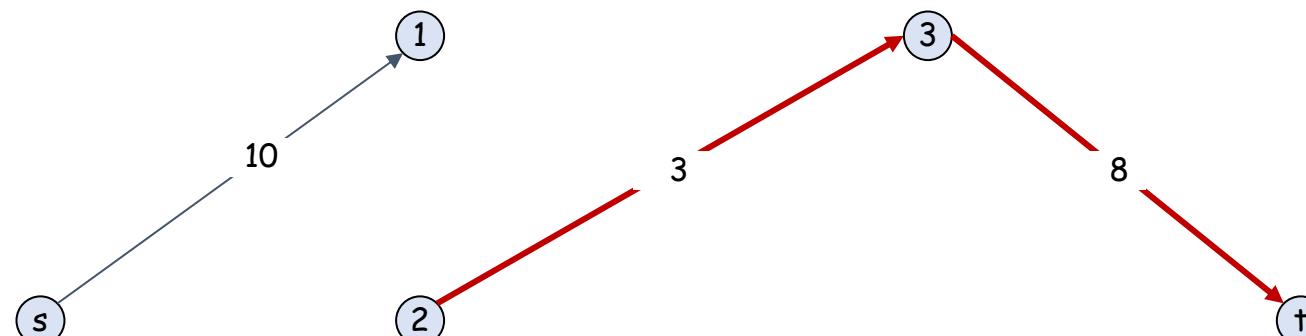


Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
- If get stuck, delete node from L_G and retreat to previous node.

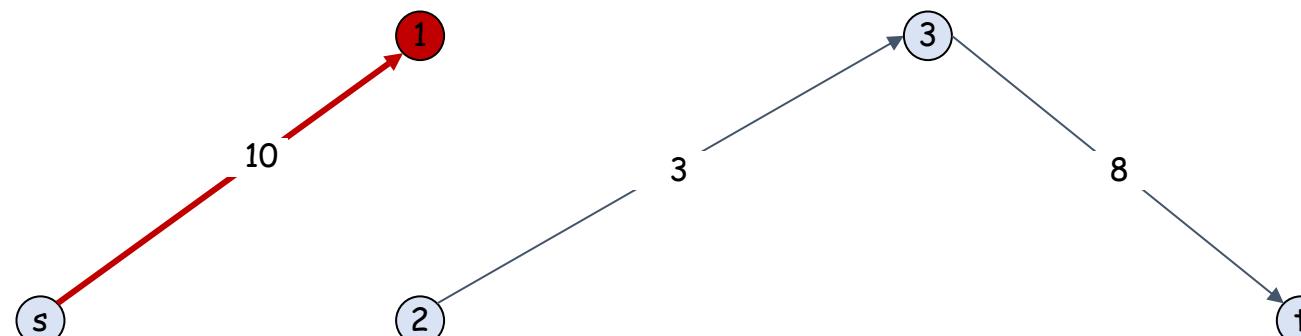


Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

- **Phase of normal augmentations:**

- Construct level graph L_G .
- Start at s , advance along an edge in L_G until reach t or get stuck.
- If reach t , augment flow; update L_G ; and restart from s .
- If get stuck, delete node from L_G and retreat to previous node.

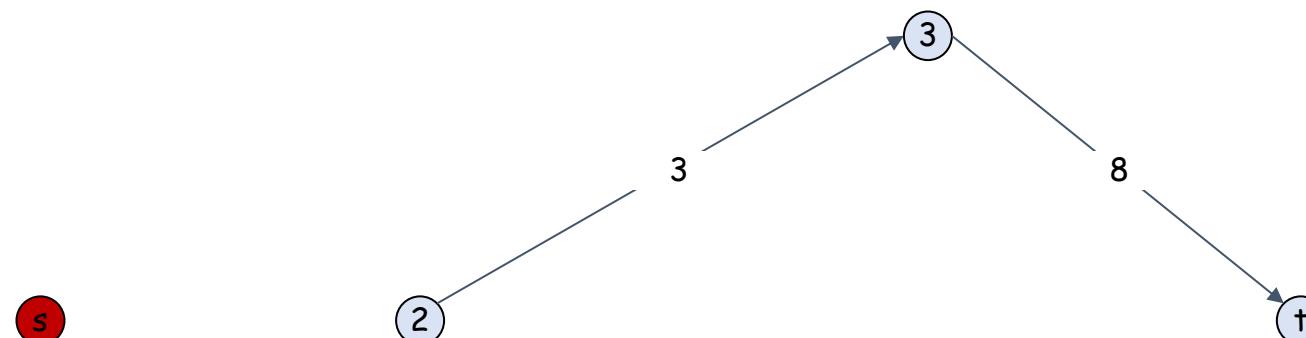




Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.

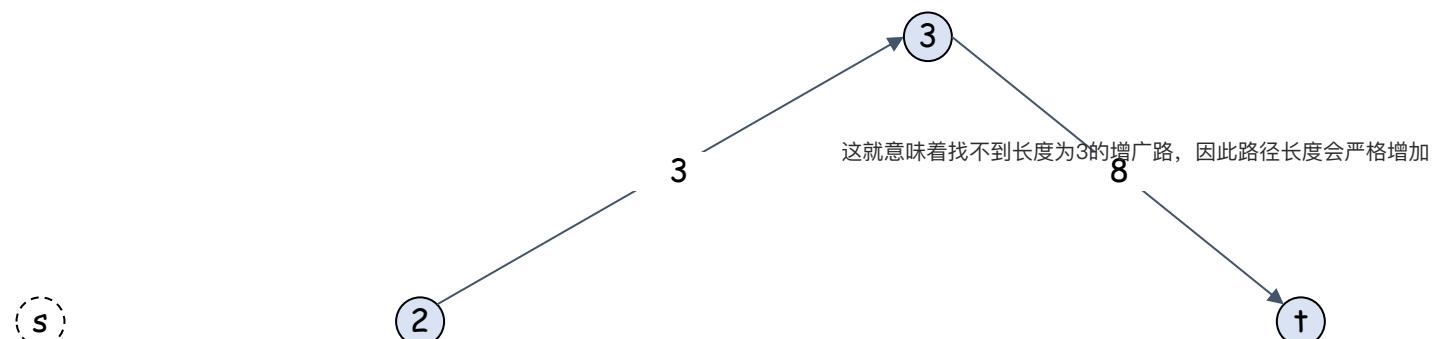
- **Phase of normal augmentations:**
 - Construct level graph L_G .
 - Start at s , advance along an edge in L_G until reach t or get stuck.
 - If reach t , augment flow; update L_G ; and restart from s .
 - If get stuck, delete node from L_G and retreat to previous node.





Dinitz's Algorithm

- **Two types of augmentations:**
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.
- **Phase of normal augmentations:**
 - Construct level graph L_G .
 - Start at s , advance along an edge in L_G until reach t or get stuck.
 - If reach t , augment flow; update L_G ; and restart from s .
 - If get stuck, delete node from L_G and retreat to previous node.





Dinitz's Algorithm

- Two types of augmentations:
 - Normal: length of shortest path does not change.
 - Special: length of shortest path strictly increases.
- Dinitz's algorithm per normal phase: (as refined by Even and Itai)

```
Dinitz-Normal-Phase( $G_f$ ,  $s$ ,  $t$ ) {  
     $L_G$  = level graph of  $G_f$   
     $P$  = empty path  
    Advance( $s$ )  
}  
    optimization: can instead advance  
    from the nearest advancable node →  
    Retreat( $v$ ) {  
        if ( $v = s$ ) return  
        else  
            delete  $v$  and incident edges from  $L_G$   
            remove last edge  $(u, v)$  from  $P$   
            Advance( $u$ )  
    }  
  
    Advance( $v$ ) {  
        if ( $v = t$ )  
             $f = \text{Augment}(f, c, P)$   
            remove bottleneck edges from  $L_G$   
             $P$  = empty path  
            Advance( $s$ )  
        if (there exists  $(v, w) \in L_G$ )  
            add edge  $(v, w)$  to  $P$   
            Advance( $w$ )  
        Retreat( $v$ )  
    }
```



Dinitz's Algorithm

- **Two types of augmentations:**

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

- **Dinitz's algorithm:**

```
Dinitz(G, s, t, c) {
    foreach e ∈ E: f(e) = 0
    Gf = residual network of G with respect to flow f
    上面是special case, 重建level graph
    while (there exists an augmenting path P in Gf) {
        Dinitz-Normal-Phase(Gf, s, t)
    }
    return f
}
```

bfs做个level graph, 判断能不能到达t



Dinitz's Algorithm: Analysis

- **Lemma.** A phase can be implemented to run in $O(mn)$ time.
- **Pf. (direct proof)** 要 $O(1)$ 时间确定边在原图中的位置
 - Level graph initialization happens once per phase. $\leftarrow O(m)$ per phase using BFS
 - At most m augmentations (updating f) per phase. $\leftarrow O(mn)$ per phase 每条增广路最多做 $O(n)$ 次更新
(because an augmentation deletes at least one edge from L_G)
 - At most n retreats per phase. $\leftarrow O(m + n)$ per phase
(because a retreat deletes one node and all incident edges from L_G)
 - At most mn advances per phase. $\leftarrow O(mn)$ per phase
(because at most n advances before retreat or augmentation) ▪
- **Theorem.** [Dinitz 1970] Dinitz's algorithm runs in $O(mn^2)$ time.
- **Pf.** There are at most $n - 1$ phases and each phase runs in $O(mn)$ time. ▪



Augmenting-Path Algorithms: Summary

year	method	# augmentations	running time	
1955	augmenting path	$n C$	$O(m n C)$	C 是所有流量的最大值
1972	fattest path	$m \log(mC)$	$O(m^2 \log n \log(mC))$	找流量大的增广路
1972	capacity scaling	$m \log C$	$O(m^2 \log C)$	fat paths
1985	improved capacity scaling	$m \log C$	$O(m n \log C)$	
1970	shortest augmenting path	$m n$	$O(m^2 n)$	长度最短的增广路
1970	level graph	$m n$	$O(m n^2)$	shortest paths
1983	dynamic trees	$m n$	$O(m n \log n)$	

augmenting-path algorithms with m edges, n nodes, and integer capacities between 1 and C

具体看图的大小、 C 的大小来选择



Max-Flow Algorithms: Theory Highlights

year	method	worst case	discovered by
1951	simplex	$O(m n^2 C)$	Dantzig
1955	augmenting paths	$O(m n C)$	Ford–Fulkerson
1970	shortest augmenting paths	$O(m n^2)$	Edmonds–Karp, Dinitz
1974	blocking flows	$O(n^3)$	Karzanov
1983	dynamic trees	$O(m n \log n)$	Sleator–Tarjan
1985	improved capacity scaling	$O(m n \log C)$	Gabow
1988	push-relabel 实际用的多	$O(m n \log (n^2 / m))$	Goldberg–Tarjan
1998	binary blocking flows	$O(m^{3/2} \log (n^2 / m) \log C)$	Goldberg–Rao
2013	compact networks	$O(m n)$	Orlin
2014	interior-point methods	$\tilde{O}(m n^{1/2} \log C)$	Lee–Sidford
2016	electrical flows	$\tilde{O}(m^{10/7} C^{1/7})$	Mądry
20xx		???	

← best in practice

max-flow algorithms with m edges, n nodes, and integer capacities between 1 and C



Max-Flow Algorithms: Practice

- **Caveat.** **Worst-case** running time is generally **not useful** for predicting or comparing max-flow algorithm performance **in practice**.
- **Best in practice.** Push-relabel algorithm [Goldberg-Tarjan 1988] with gap relabeling: $O(n^2m^{1/2})$ in practice. [section 7.4 of textbook]
 - Increases flow one edge at a time instead of one augmenting path at a time.
- **Computer vision.** Different algorithms work better for some **dense** problems that arise in applications to computer vision.
- **Implementation.** MATLAB, Google **OR-Tools**, etc.



Announcement

- Lab 11 will be released today and the deadline is May 20.

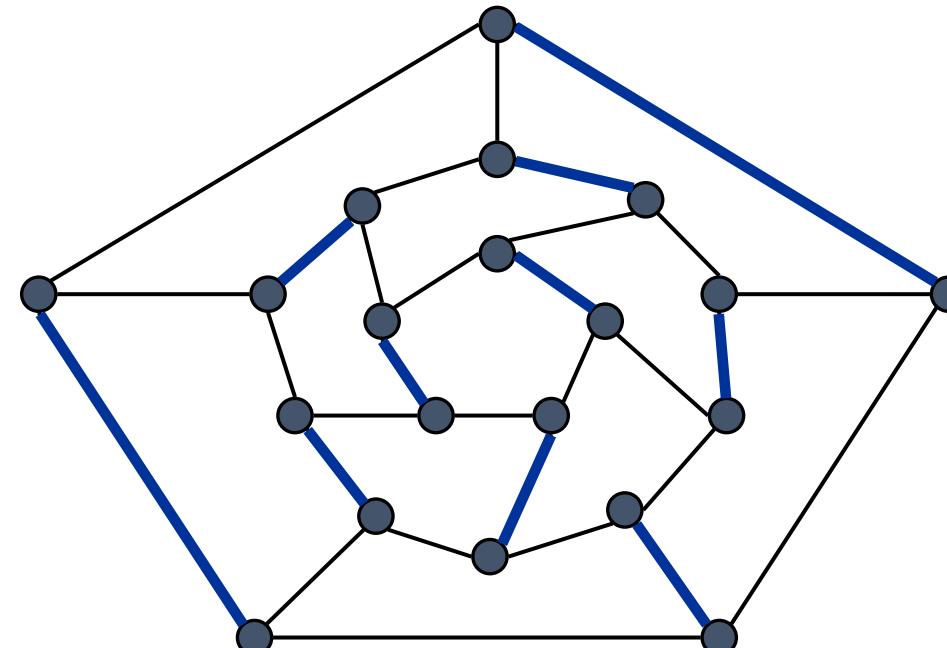


南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

7. Bipartite Matching

Matching

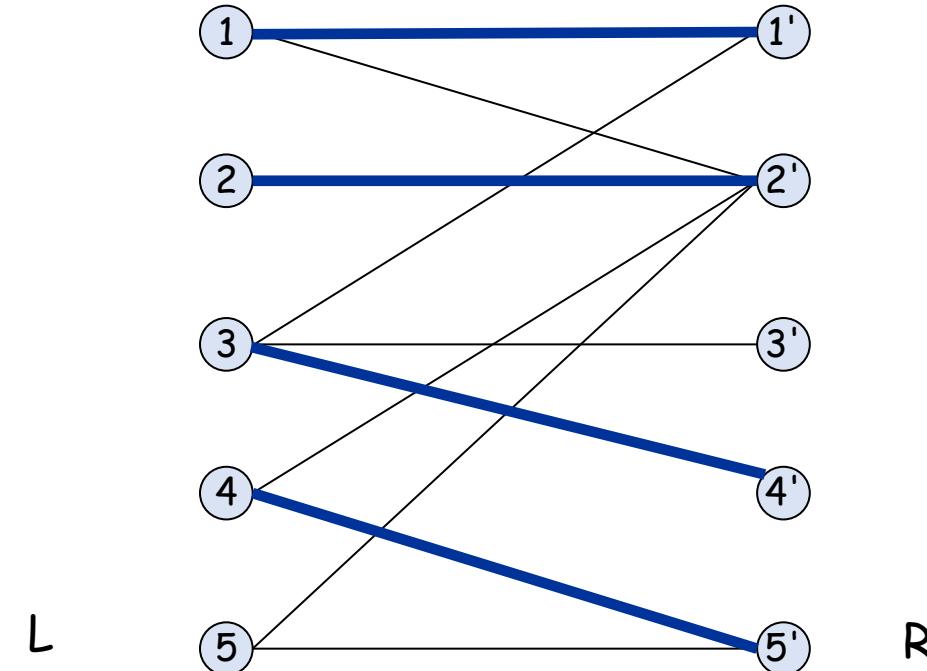
- **Def.** Given an undirected graph $G = (V, E)$, the subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M .
- **Maximum Matching Problem.** Find max-cardinality matching.





Bipartite Matching

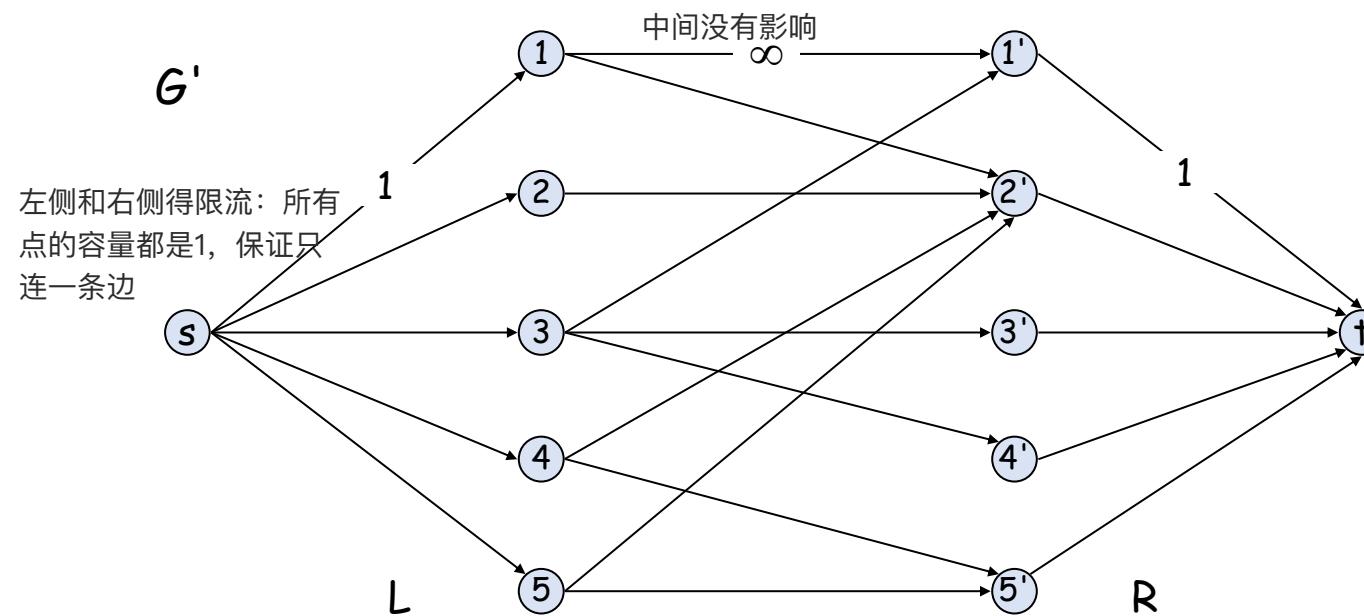
- **Def.** Given an undirected bipartite graph $G = (L \cup R, E)$, the subset of edges $M \subseteq E$ is a **matching** if each node appears in ≤ 1 edge in M .
- **Maximum Bipartite Matching Problem.** Find **max-cardinality** matching in a bipartite graph.



Bipartite Matching: Max-Flow Formulation

- **Max-flow formulation:**

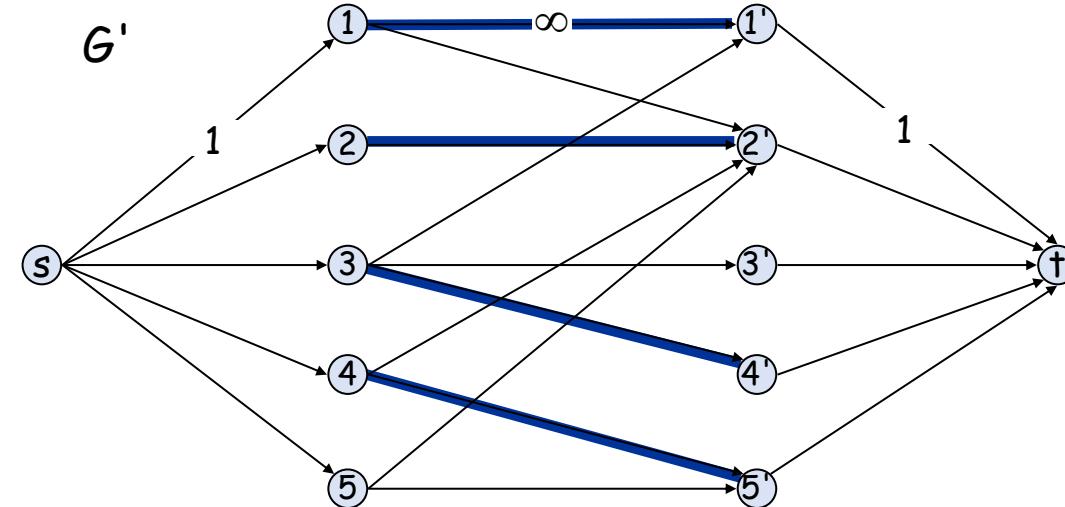
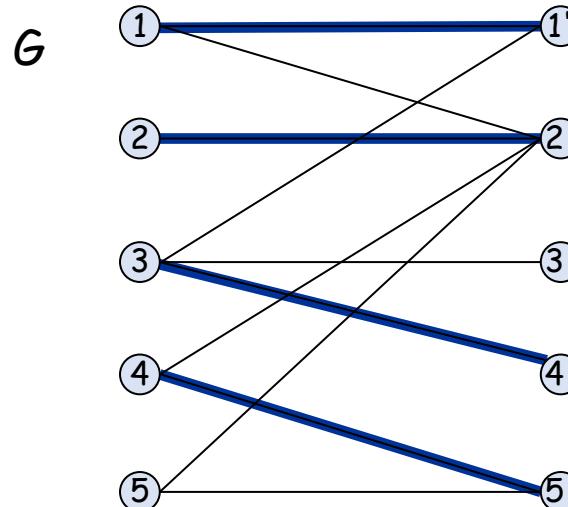
- Create a directed graph $G' = (L \cup R \cup \{s, t\}, E')$.
- Direct all edges from L to R , and assign **infinite** (or **unit**) capacity.
- Add source s , and **unit-capacity** edges from s to each node in L .
- Add sink t , and **unit-capacity** edges from each node in R to t .





Max-Flow Formulation: Correctness

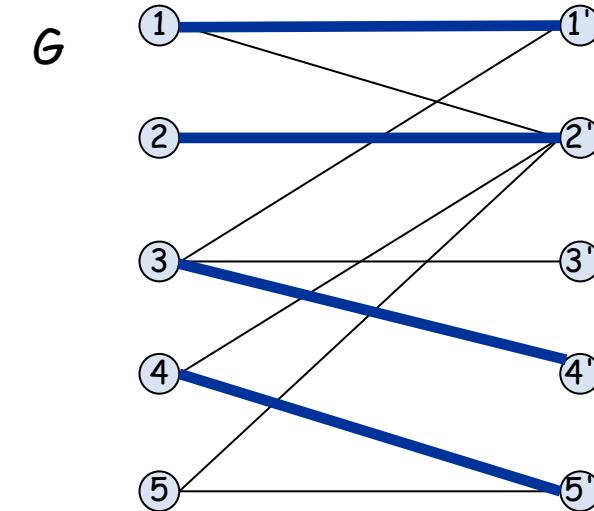
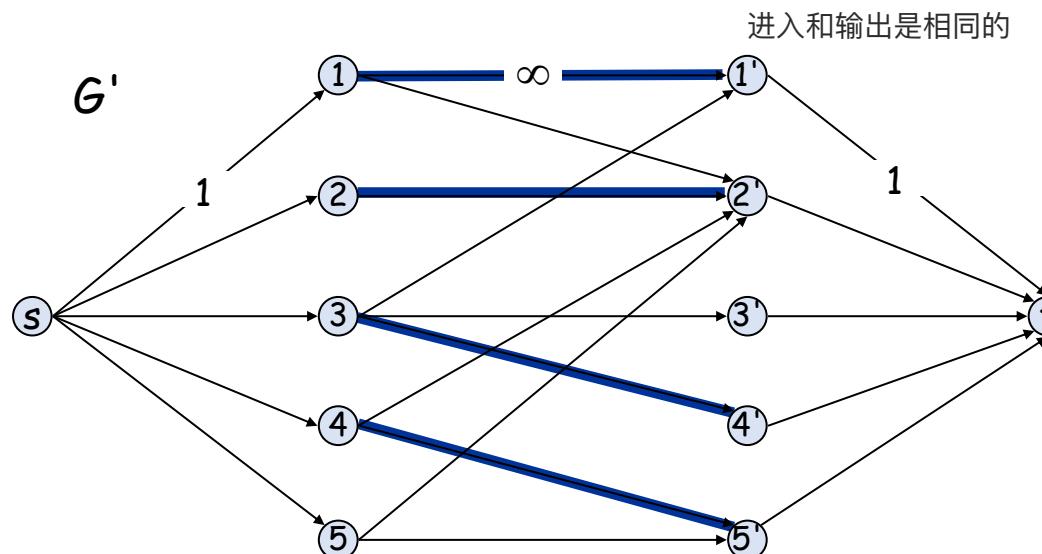
- **Theorem.** There exists a **1-1 correspondence** between matchings of cardinality k in G and integral flows of value k in G' .
- **Pf. $\Rightarrow:$** Let M be a matching in G of cardinality k .
 - Consider flow f that sends **1** unit flow on each of the k corresponding paths.
 - f is an integral flow of value k . ▀





Max-Flow Formulation: Correctness

- **Theorem.** There exists a **1-1 correspondence** between matchings of cardinality k in G and integral flows of value k in G' .
- **Pf.** \Leftarrow : Let f be an integral flow in G' of value k .
 - Consider $M = \text{set of edges from } L \text{ to } R \text{ with } f(e) = 1$:
 - ✓ Each node in L and R participates in **at most one edge** in M .
 - ✓ $|M| = k$: apply **Flow Value Lemma** to cut $(L \cup \{s\}, R \cup \{t\})$. ▀





Max-Flow Formulation: Correctness

- **Theorem.** There exists a **1-1 correspondence** between matchings of cardinality k in G and integral flows of value k in G' .
- **Corollary.** Can find **max bipartite matching** via max-flow formulation.
- **Pf. (direct proof)**
 - **Integrality Theorem** \Rightarrow there exists a **max flow f^*** in G' that is **integral**.
 - **Theorem** $\Rightarrow f^*$ corresponds to **max-cardinality matching**. ▀



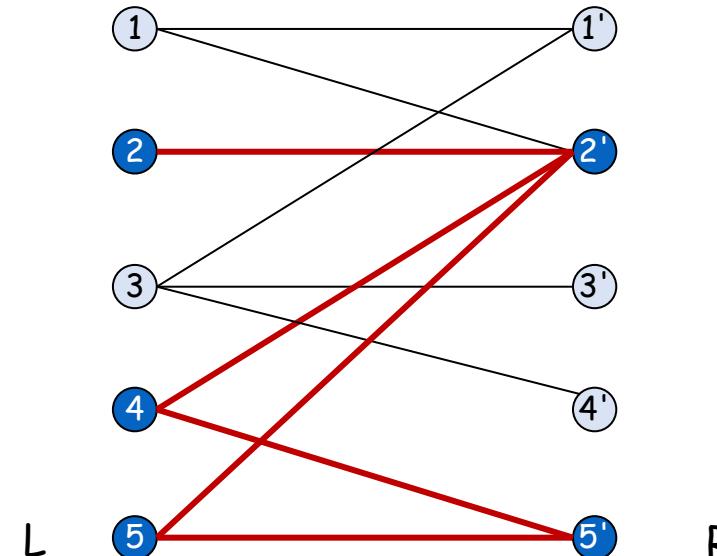
Perfect Matchings in Bipartite Graphs

- **Def.** Given a graph $G = (V, E)$, a subset of edges $M \subseteq E$ is a **perfect matching** if each node appears in **exactly one** edge in M .
- **Q.** When does a bipartite graph have a **perfect matching**?
- **Structure of bipartite graphs with perfect matchings:**
 - Clearly we must have $|L| = |R|$.
 - What other conditions are **necessary**?
 - What other conditions are **sufficient**?



Perfect Matchings in Bipartite Graphs

- **Notation.** Let S be a subset of nodes, and let $N(S)$ be the set of nodes adjacent to nodes in S .
- **Claim.** If a bipartite graph $G = (L \cup R, E)$ has a perfect matching, then $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.
- **Pf.** Each node in S has to be matched to a different node in $N(S)$.
- **Example:**



No perfect matching exists:

$$S = \{ 2, 4, 5 \}$$

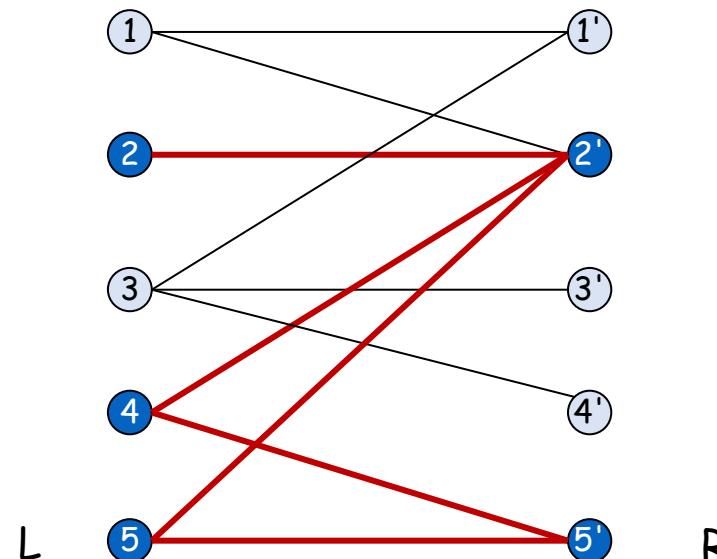
$$N(S) = \{ 2', 5' \}.$$



Hall's Marriage Theorem

- **Theorem.** [Frobenius 1917, Hall 1935] Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. G has a perfect matching if and only if $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.
- **Pf.** “only if” \Rightarrow : This was the previous **Claim**.

如果不成立的话，那么就左多，右少，不可能匹配了



No perfect matching exists:

$$S = \{ 2, 4, 5 \}$$

$$N(S) = \{ 2', 5' \}.$$

Hall's Marriage Theorem

- Pf. “if” \Leftarrow : Suppose G does **not** have a **perfect matching**. (contrapositive)
 - Formulate as a max flow problem and let (A, B) be a **min cut** in G' .
 - **Max-Flow Min-Cut Theorem** $\Rightarrow c(A, B) < |L|$ 最大流小于顶点个数, 不然就有完全匹配了
 - Define $L_A = L \cap A$, $L_B = L \cap B$, $R_A = R \cap A$.
 - Min cut cannot use ∞ edges $\Rightarrow N(L_A) \subseteq R \cap A = R_A$ 建模后可以把L和R之间的边权都设为正无穷
 - $c(A, B) = |L_B| + |R_A| < |L| = |L_A| + |L_B| \Rightarrow |R_A| < |L_A|$ 最小切一定切不到无穷边权
 - Together, $|N(L_A)| \leq |R_A| < |L_A|$ and $L_A \subseteq L$.
 - The contrapositive is true. ▀

所有 L_A 的邻接点 $N(L_A)$ 都必须落在 R_A 中 (因为这些点不能连接到 $R \setminus A$)

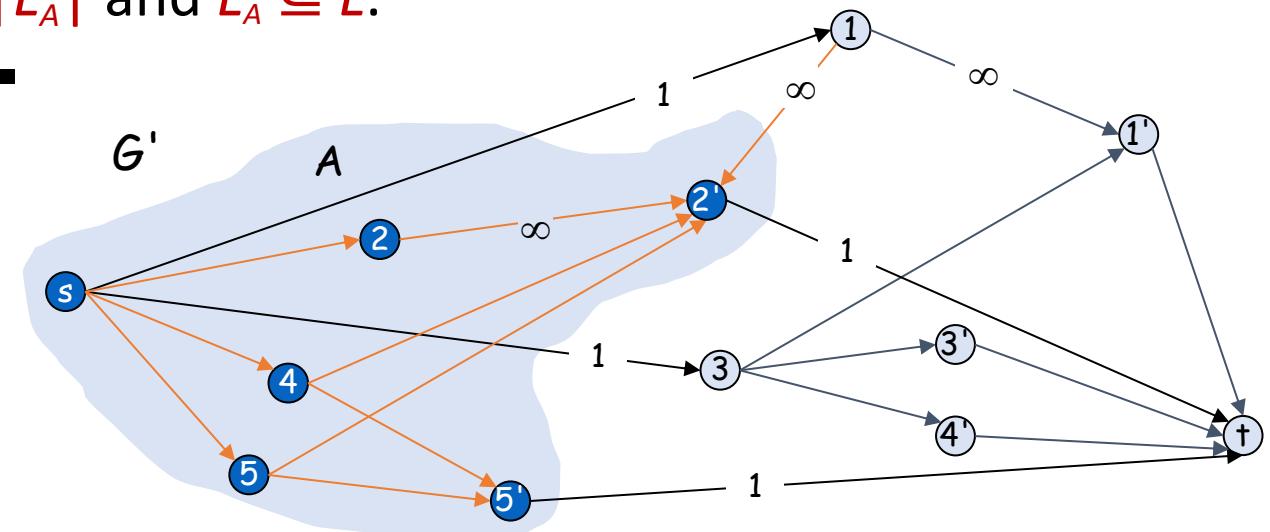
$$\Rightarrow N(L_A) \subseteq R_A \Rightarrow |N(L_A)| \leq |R_A|$$

$$L_A = \{2, 4, 5\}$$

$$L_B = \{1, 3\}$$

$$R_A = \{2', 5'\}$$

$$N(L_A) = \{2', 5'\}$$





Max Matching: Closing Remarks

- **Algorithms for Bipartite Matching:**

- Edmonds-Karp: $O(m^2n)$
- Capacity-scaling: $O(m^2 \log C) = O(m^2)$
- Ford-Fulkerson: $O(m v(f^*)) = O(mn)$
- Dinitz: $O(mn^{1/2})$ [Even-Tarjan 1975]
- Fast matrix multiplication: $O(n^{2.378})$ [Mucha-Sankowski 2003]

learned max-flow algorithms

- **Non-Bipartite Matching:**

- Structure of non-bipartite (undirected) graphs is more complicated.
- But well-understood. [Tutte-Berge formula, Edmonds-Galai]
- Blossom algorithm: $O(n^4)$ [Edmonds 1965]
- Best known: $O(mn^{1/2})$ [Micali-Vazirani 1980, Vazirani 1994]



Historical Significance (Jack Edmonds 1965)

2. Digression. An explanation is due on the use of the words “efficient algorithm.” First, what I present is a conceptual description of an algorithm and not a particular formalized algorithm or “code.”

For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, “efficient” means “adequate in operation or performance.” This is roughly the meaning I want—in the sense that it is conceivable for maximum matching to have no efficient algorithm. Perhaps a better word is “good.”

I am claiming, as a mathematical result, the existence of a *good* algorithm for finding a maximum cardinality matching in a graph.

There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether *or not* there exists an algorithm whose difficulty increases only algebraically with the size of the graph.

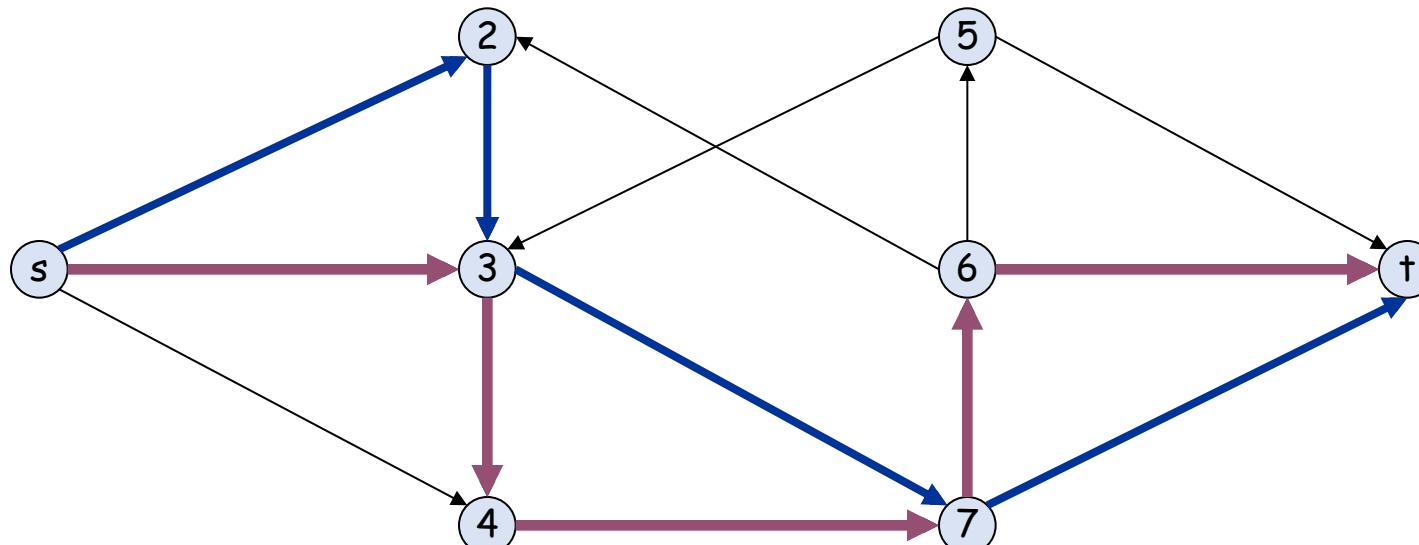


南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

8. Disjoint Paths

Edge-Disjoint Paths

- **Def.** Two paths are **edge-disjoint** if they have no edge in common.
- **Edge-disjoint paths problem.** Given a directed graph $G = (V, E)$ and two nodes s and t , find the **max** number of edge-disjoint $s-t$ paths.
- **Example.** Communication networks.

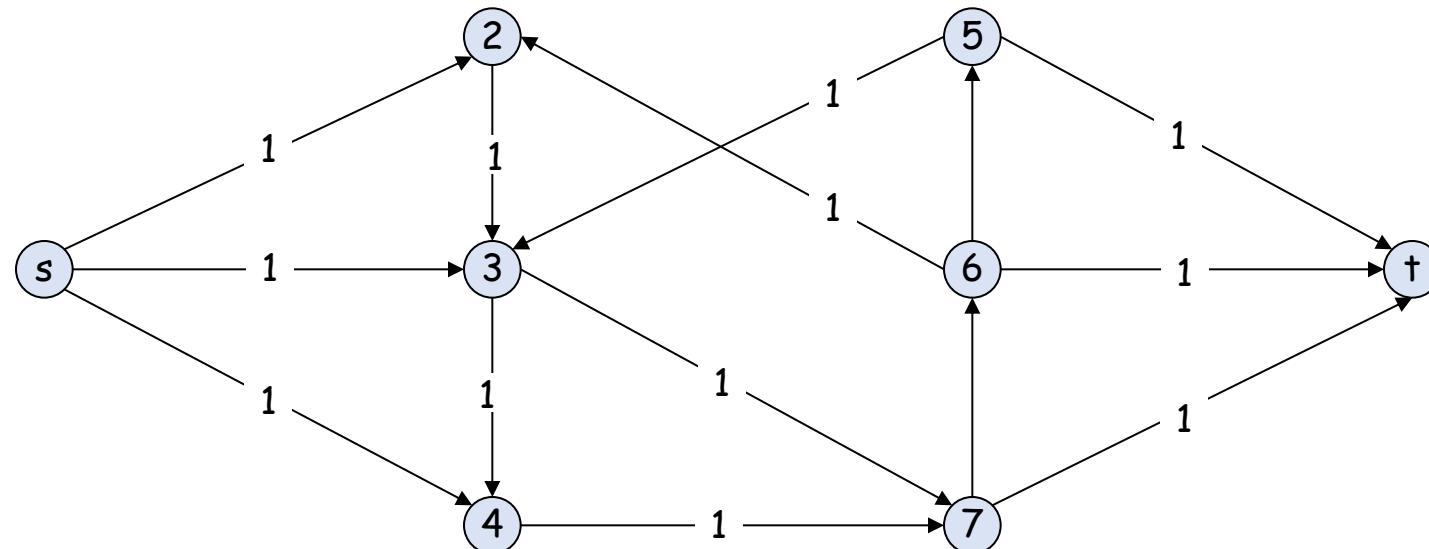




Edge-Disjoint Paths: Max-Flow Formulation

边不重复

- **Def.** Two paths are **edge-disjoint** if they have no edge in common.
- **Edge-disjoint paths problem.** Given a directed graph $G = (V, E)$ and two nodes s and t , find the **max** number of edge-disjoint $s-t$ paths.
- **Max-flow formulation.** Assign **unit capacity** to every edge.





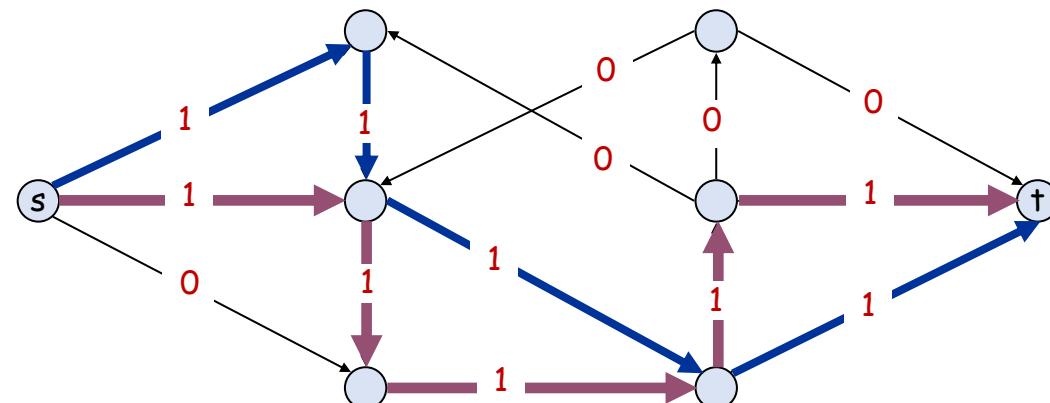
Max-Flow Formulation: Correctness

- **Theorem.** There exists a **1-1 correspondence** between k edge-disjoint $s-t$ paths in G and integral flows of value k in G' .

最大流和边互斥的解一一对应

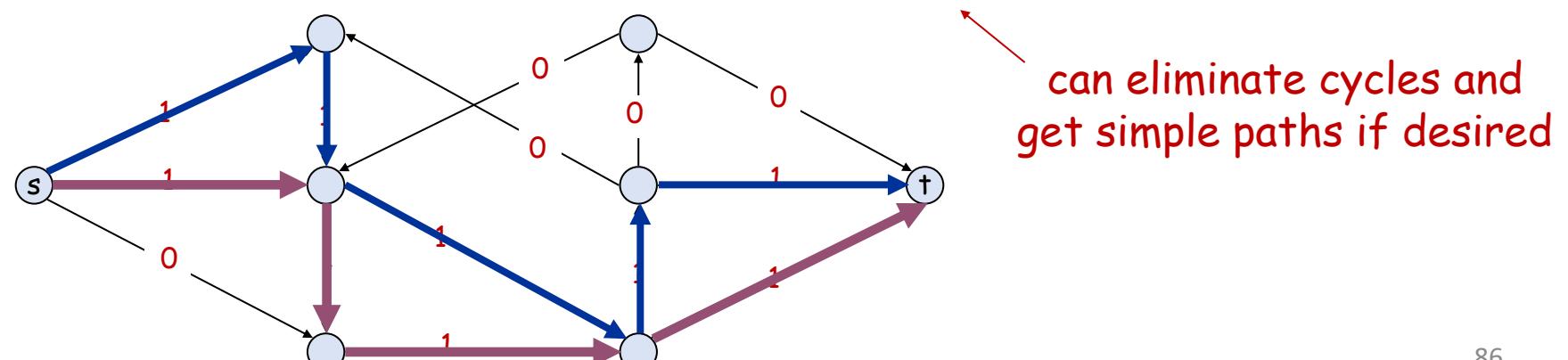
- **Pf.** $\Rightarrow:$

- Let P_1, \dots, P_k be k edge-disjoint paths in G .
- Set $f(e) = 1$ if edge e participates in some path P_i ; else set $f(e) = 0$.
- Since paths are edge-disjoint, f is a flow of value k . ▀



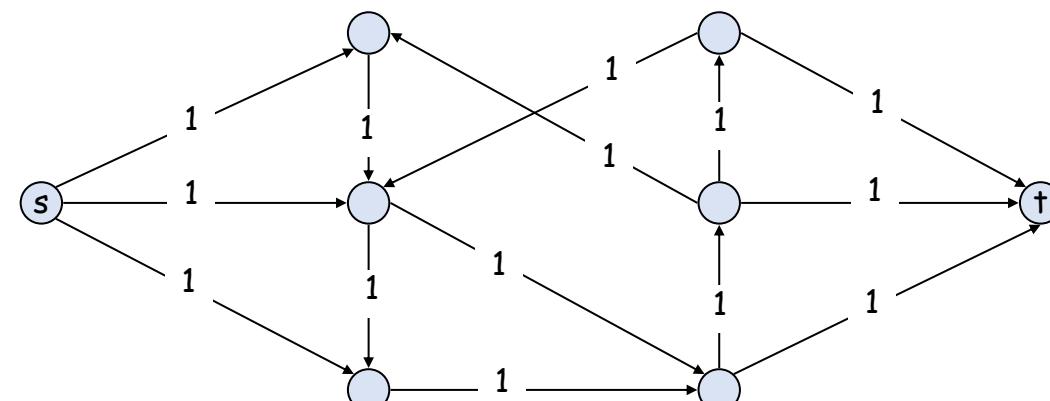
Max-Flow Formulation: Correctness

- **Theorem.** There exists a **1-1 correspondence** between k edge-disjoint $s-t$ paths in G and integral flows of value k in G' .
- **Pf. $\Leftarrow:$**
 - Let f be an integral flow in G' of value k .
 - Consider edge (s, u) with $f(s, u) = 1$:
 - ✓ **flow conservation** \Rightarrow there exists an edge (u, v) with $f(u, v) = 1$
 - ✓ continue until reach t , always choosing a **new** edge
 - Produces k edge-disjoint paths (not necessarily simple) . ▀



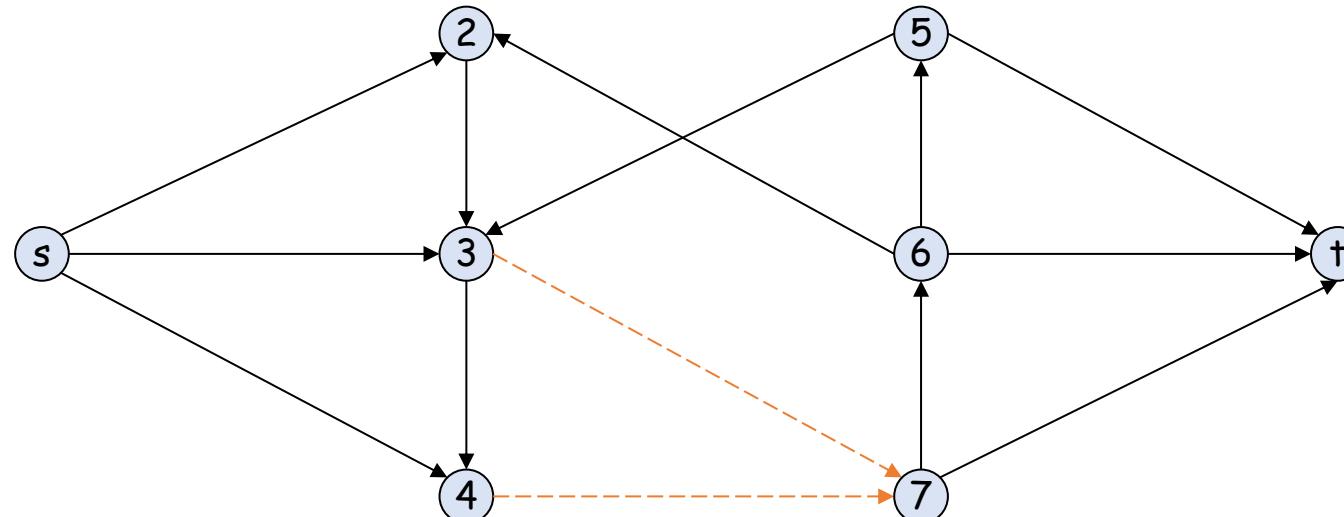
Max-Flow Formulation: Correctness

- **Max-flow formulation.** Assign **unit capacity** to every edge.
- **Theorem.** There exists a **1-1 correspondence** between **k** edge-disjoint s - t paths in G and integral flows of value **k** in G' .
- **Corollary.** Can solve edge-disjoint paths via max-flow formulation.
- **Pf. (direct proof)**
 - **Integrality theorem** \Rightarrow there exists a max flow f^* in G' that is integral.
 - **Theorem** $\Rightarrow f^*$ corresponds to **max** number of edge-disjoint s - t paths in G . ▀



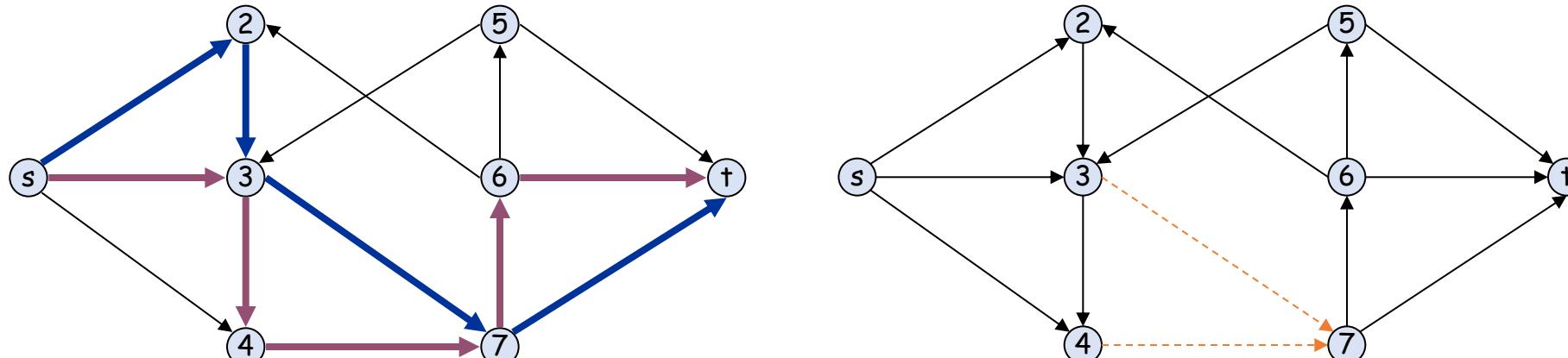
Network Connectivity

- **Def.** A set of edges $F \subseteq E$ disconnects t from s if every $s-t$ path uses at least one edge in F .
- **Network connectivity.** Given a directed graph $G = (V, E)$ and two nodes s and t , find min number of edges whose removal disconnects t from s .



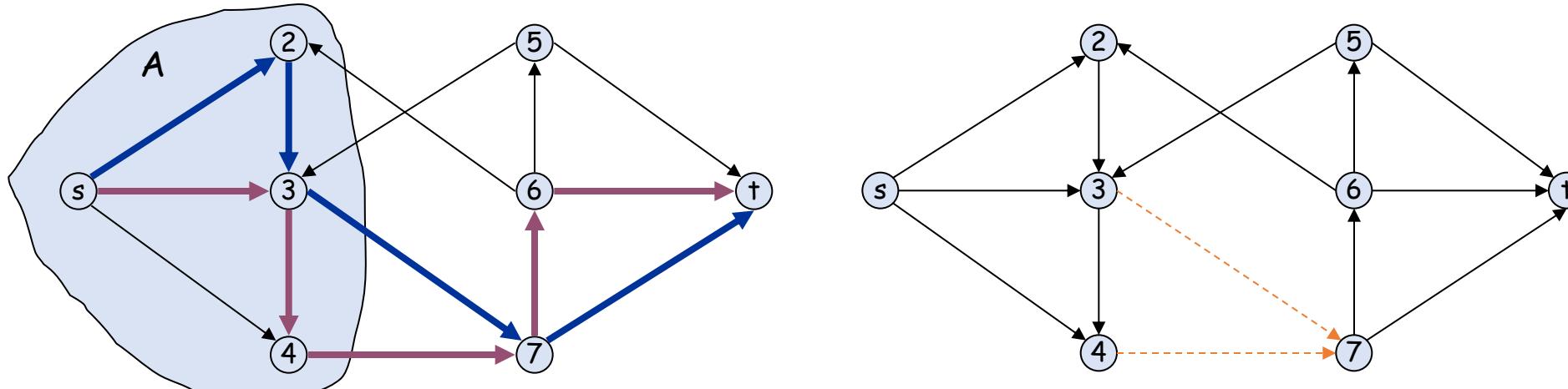
Menger's Theorem

- **Theorem.** [Menger 1927] The **max** number of **edge-disjoint $s-t$ paths** equals the **min** number of edges whose removal **disconnects t from s** .
- **Pf. \leq :**
 - Suppose the removal of $F \subseteq E$ disconnects t from s , and $|F| = k$.
 - Every $s-t$ path uses **at least** one edge in F .
 - Hence, the number of **edge-disjoint** paths is $\leq k$. ▀



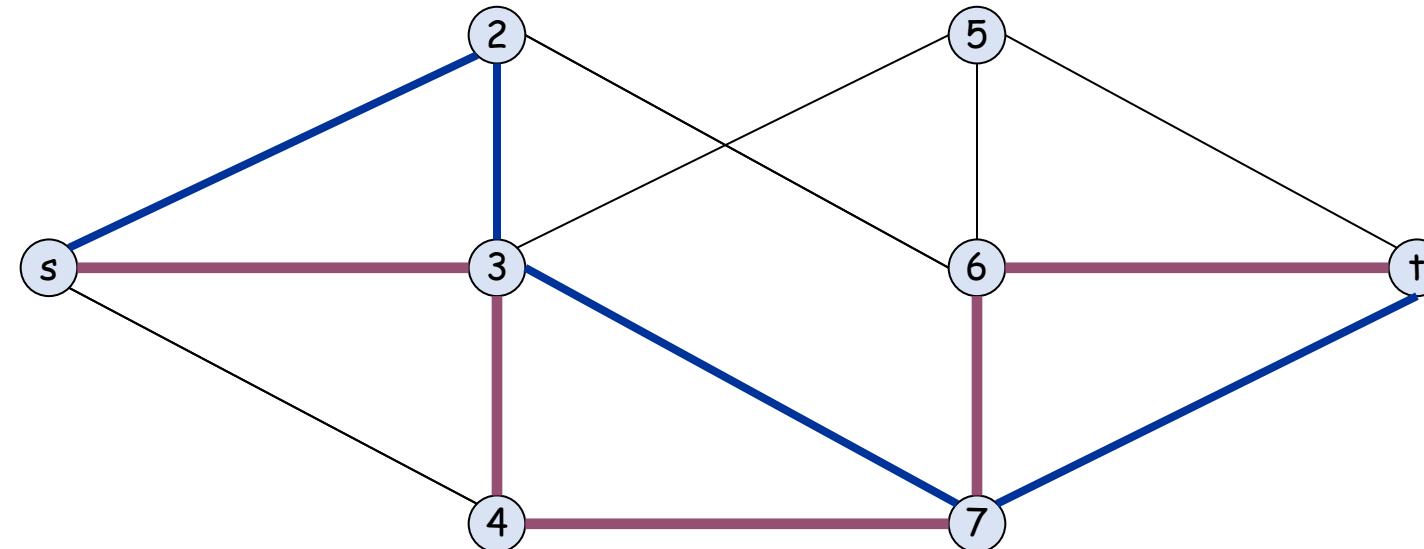
Menger's Theorem

- **Theorem.** [Menger 1927] The **max** number of **edge-disjoint $s-t$ paths** equals the **min** number of edges whose removal **disconnects t from s** .
- **Pf. \geq :**
 - Suppose max number of edge-disjoint $s-t$ paths is k . Then, value of max flow = k .
 - **Max-flow min-cut theorem** \Rightarrow there exists a cut (A, B) of capacity k .
 - Let F be set of edges going from A to B .
 - $|F| = k$ and removing F disconnects t from s . ▀



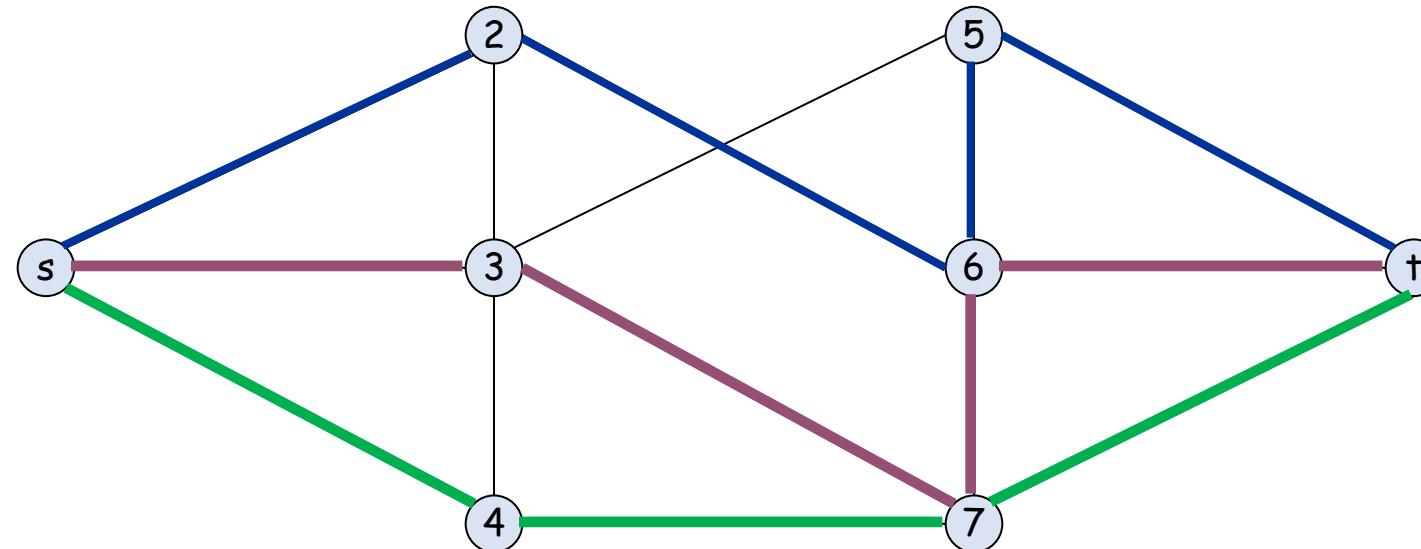
Edge-Disjoint Paths in Undirected Graphs

- **Edge-disjoint paths problem in undirected graphs.** Given an **undirected** graph $G = (V, E)$ and two nodes s and t , find **max** number of edge-disjoint s - t paths.
- **Example.** 2 edge-disjoint paths.



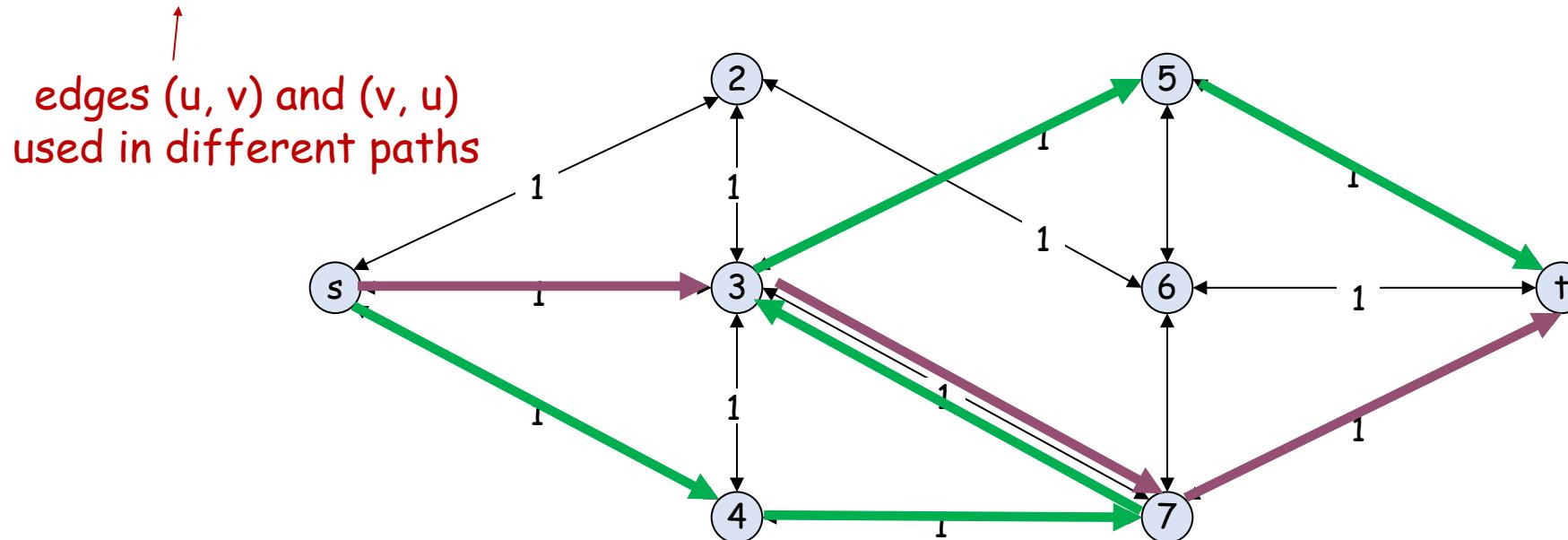
Edge-Disjoint Paths in Undirected Graphs

- **Edge-disjoint paths problem in undirected graphs.** Given an **undirected** graph $G = (V, E)$ and two nodes s and t , find **max** number of edge-disjoint s - t paths.
- **Example.** 3 edge-disjoint paths (max number).



Edge-Disjoint Paths: Max-Flow Formulation

- **Max-flow formulation.** Replace each edge with two antiparallel edges and assign **unit capacity** to every edge.
- **Caveat.** Two paths P_1 and P_2 may be edge-disjoint in the **directed** graph but **not** edge-disjoint in the **undirected** graph.





Max-Flow Formulation: Correctness

- **Max-flow formulation.** Replace each edge with two antiparallel edges and assign unit capacity to every edge.
- **Lemma.** In any flow network, there exists a maximum flow f in which for each pair of antiparallel edges e and e' : $f(e) = 0$ or $f(e') = 0$ or both. Moreover, **integrality theorem** still holds.
防止边被用两次，把重复边变没
- **Pf.** (by induction on the number of such edge pairs)
 - Suppose $f(e) > 0$ and $f(e') > 0$ for a pair of antiparallel edges e and e' .
 - Set $f'(e) = f(e) - \delta$ and $f'(e') = f(e') - \delta$, where $\delta = \min\{f(e), f(e')\}$.
 - f' is still a flow of the same value but has one fewer such pair. ▀



Max-Flow Formulation: Correctness

- **Max-flow formulation.** Replace each edge with two antiparallel edges and assign unit capacity to every edge.
- **Lemma.** In any flow network, there exists a maximum flow f in which for each pair of antiparallel edges e and e' : $f(e) = 0$ or $f(e') = 0$ or both. Moreover, **integrality theorem** still holds.
- **Theorem.** Max number of edge-disjoint $s-t$ paths = value of max flow.
- **Pf.** Similar to the proof in directed graphs. Use the above **Lemma**.



More Menger Theorems

- **Theorem.** Given an **undirected** graph and two nodes s and t , the **max** number of **edge-disjoint s - t** paths equals the **min** number of edges whose removal **disconnects s and t** .
- **Theorem.** Given an **undirected** graph and two nonadjacent nodes s and t , the **max** number of internally node-disjoint s - t paths equals the **min** number of internal nodes whose removal **disconnects s and t** .
相当于删点
- **Theorem.** Given a **directed** graph with two **nonadjacent** nodes s and t , the **max** number of internally node-disjoint s - t paths equals the **min** number of internal nodes whose removal **disconnects t from s** .



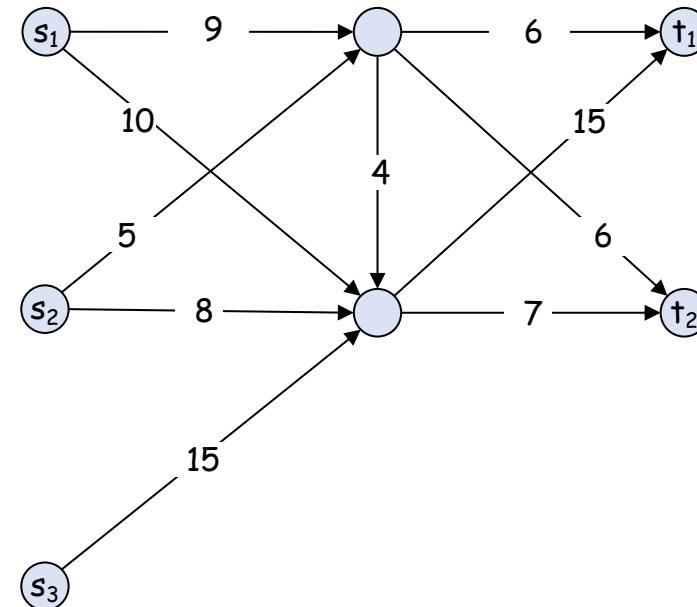
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

9. Extensions to Max Flow

Multiple Sources and Sinks

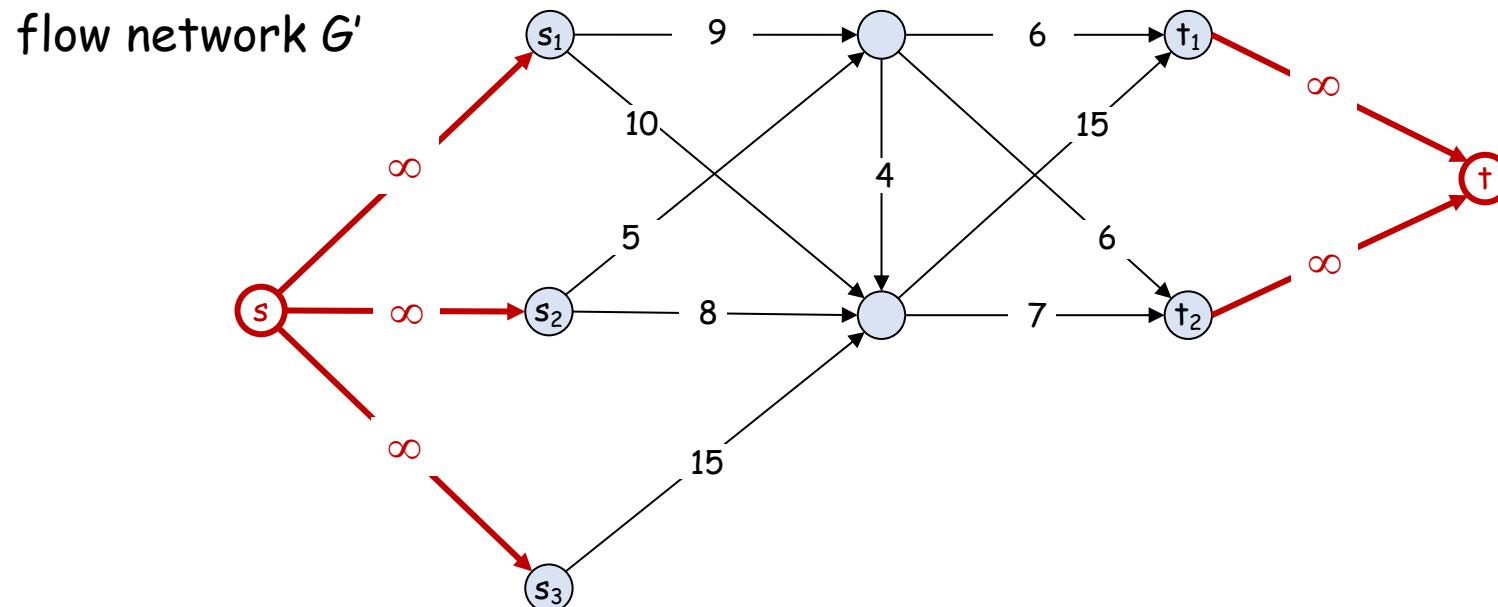
- **Def.** Given a directed graph $G = (V, E)$ with edge capacities $c(e) \geq 0$ and **multiple source nodes** and **multiple sink nodes**, find a **max flow** that can be sent from the source nodes to the sink nodes.

flow network G



Multiple Sources and Sinks

- **Def.** Given a directed graph $G = (V, E)$ with edge capacities $c(e) \geq 0$ and multiple source nodes and multiple sink nodes, find a max flow that can be sent from the source nodes to the sink nodes.
- **Claim.** There exists a 1-1 correspondence between flows in G and G' .



Circulation with Supplies and Demands

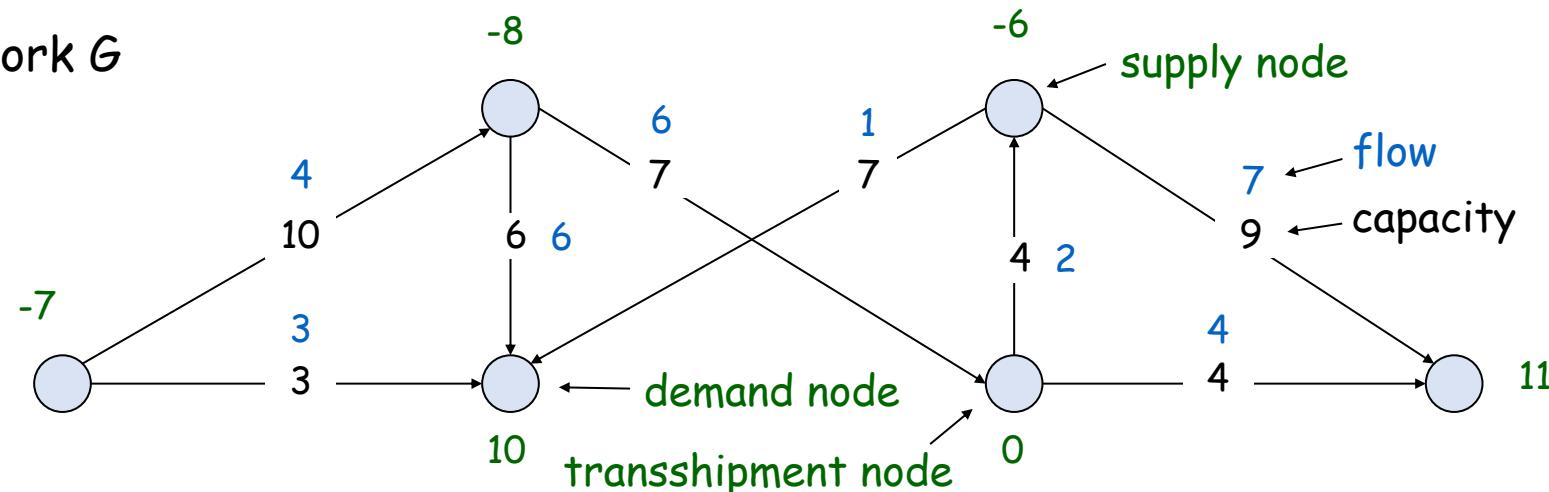
- **Def.** Given a directed graph $G = (V, E)$ with edge capacities $c(e) \geq 0$ and node demands $d(v)$, a circulation is a function $f(e)$ that satisfies:
 - [Capacity] For each $e \in E$: $0 \leq f(e) \leq c(e)$
 - [Flow conservation] For each $v \in V$: $\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$
- **Circulation problem.** Given (V, E, c, d) , find a circulation.

正值吸收, 负值释放

让他的输入和输出平衡

↑
simple notation:
 $f^{\text{in}}(v) - f^{\text{out}}(v) = d(v)$

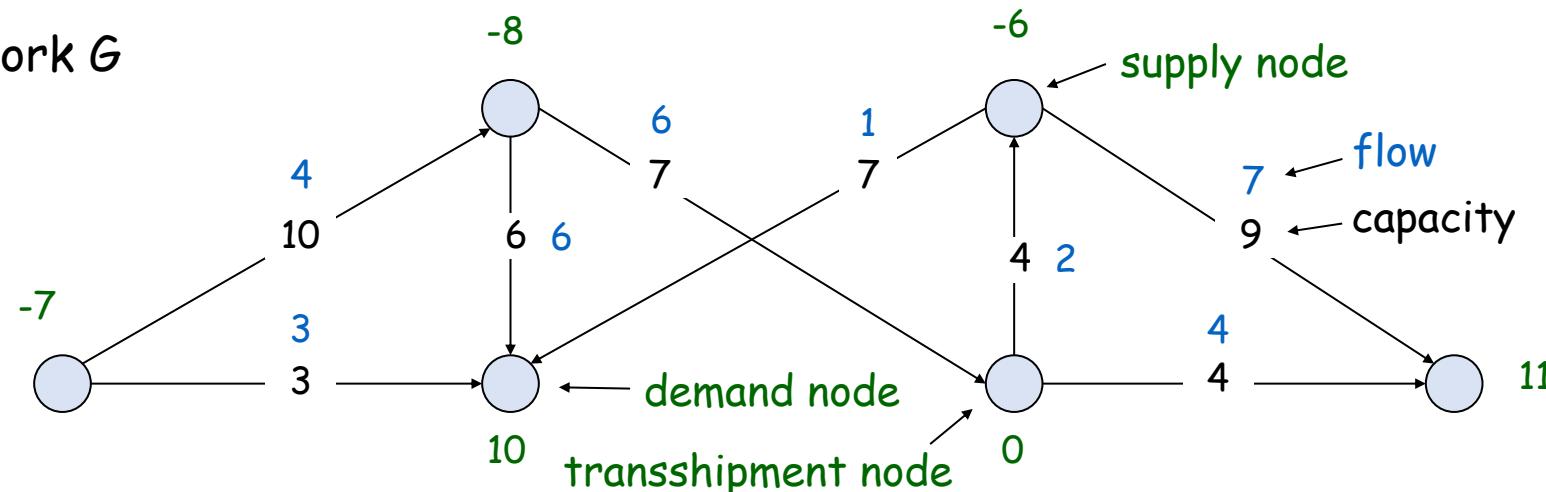
flow network G



Circulation with Supplies and Demands

- **Circulation problem.** Given (V, E, c, d) , find a circulation.
- **Observation.** G has a circulation $\Rightarrow \sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$
- **Pf.** (equivalent to show $\sum_v d(v) = 0$) total demand = total supply
 - Sum flow conservation over all $v \in V$: $\sum_v f^{in}(v) - \sum_v f^{out}(v) = \sum_v d(v)$
 - Each $f(e)$ appears once in $\sum_v f^{in}(v)$ and once in $\sum_v f^{out}(v)$ so cancel out.
 - Therefore, $\sum_v d(v) = 0$.

flow network G



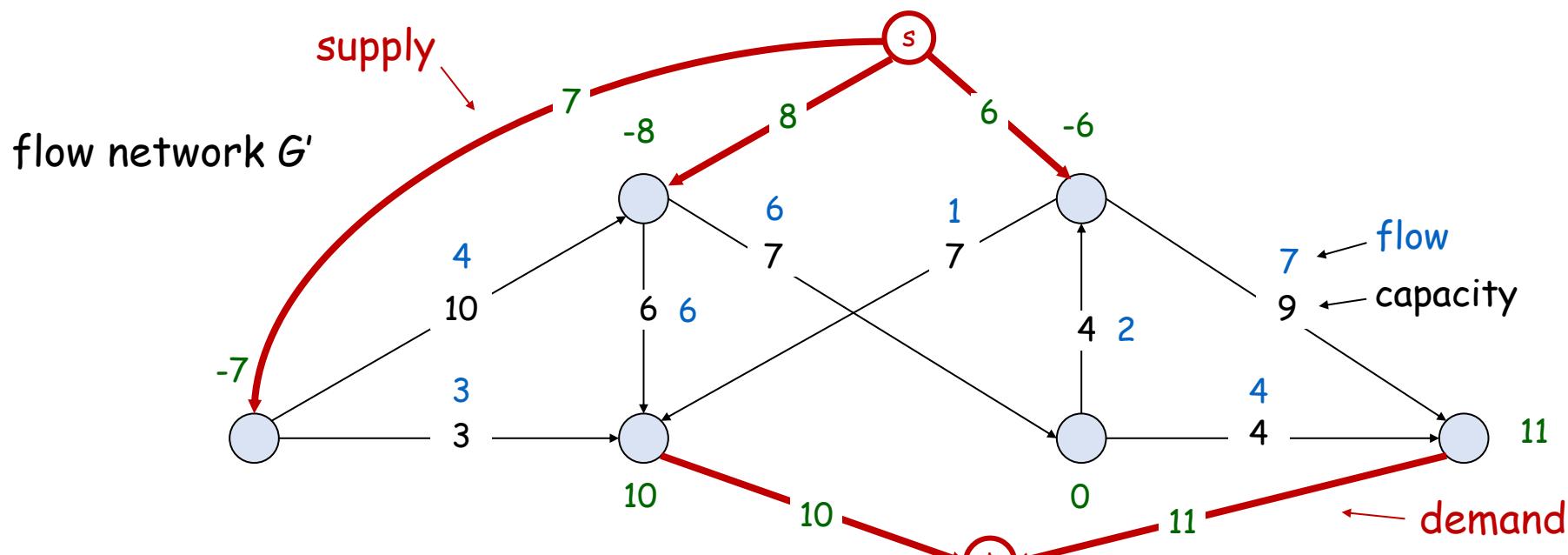


Circulation: Max-Flow Formulation

任何circulation的问题都可以转化成最大流

- **Max-flow formulation:**

- Add new source s and sink t .
- For each v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$.
- For each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$.

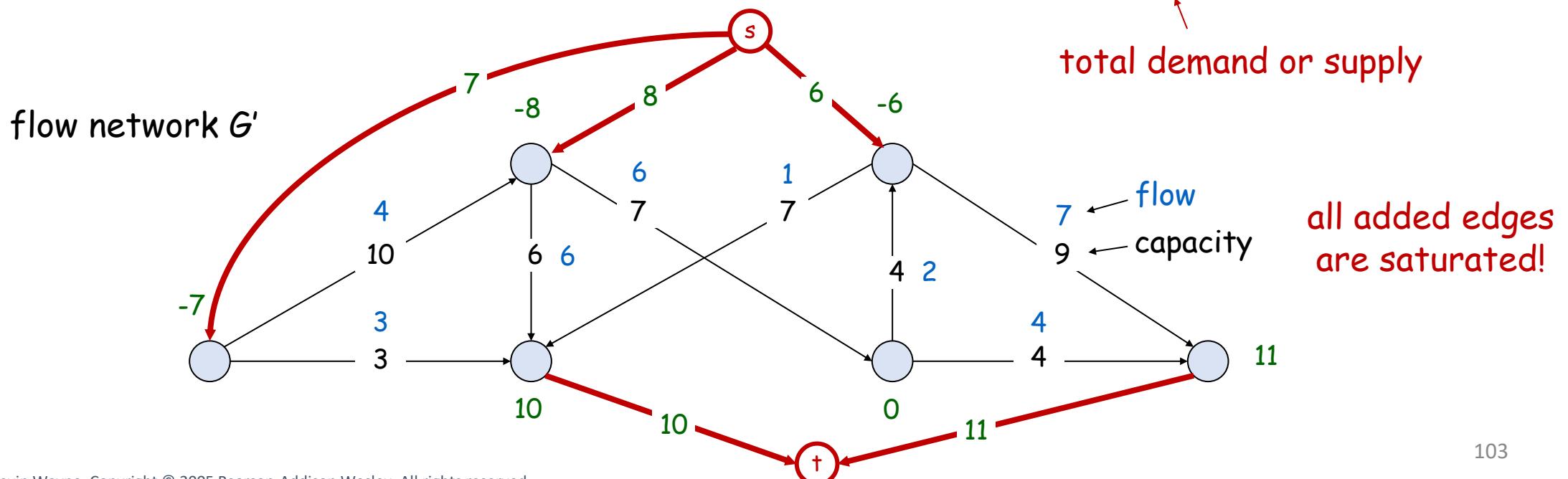


Circulation: Max-Flow Formulation

- **Max-flow formulation:**

- Add new source s and sink t .
- For each v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$.
- For each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$.

- **Claim.** G has a circulation iff G' has max flow of value $D = \sum_{v: d(v) > 0} d(v)$.





Circulation with Supplies and Demands

- **Integrality theorem.** If all capacities and demands are integers, and there exists a circulation, then there exists one that is integer-valued.
- **Pf.** Follows from **integrality theorem** for max flow and previous **Claim**.
- **Theorem.** Given (V, E, c, d) , there does not exist a circulation iff there exists a node partition (A, B) such that $\sum_{v \in B} d(v) > c(A, B)$.
- **Pf idea.** Look at a min cut in G' .

↑
exploit the relation between
cut in G' and node partition in G

↑
demand by nodes in B exceeds
supply of nodes in B plus
max capacity of edges from A to B



Circulation with Lower Bounds

- **Def.** Given a directed graph $G = (V, E)$ with edge capacities $c(e) \geq 0$, lower bounds $\ell(e) \geq 0$, and node demands $d(v)$, a **circulation** is a function $f(e)$ that satisfies:
 - [Capacity] For each $e \in E$: $\ell(e) \leq f(e) \leq c(e)$
 - [Flow conservation] For each $v \in V$: $\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$
- **Circulation problem with lower bounds.** Given (V, E, ℓ, c, d) , does there exist a feasible circulation?



Circulation with Lower Bounds

- **Max-low formulation.** Model **lower bounds** as circulation with demands.
 - Send $\ell(e)$ units of flow along edge e . ← this flow can be abstracted away from G'
 - Update demands of both endpoints.



只需要调整demand的值

- **Theorem.** There exists a circulation in G iff there exists a circulation in G' . Moreover, if all demands, capacities, and lower bounds in G are **integers**, then there exists a circulation in G that is **integer-valued**.
- **Pf idea.** $f(e)$ is a circulation in G iff $f'(e) = f(e) - \ell(e)$ is a circulation in G' .



Announcement

- Lab 12 will be released today and the deadline is May 27.



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

10. Survey Design



Survey Design

- **Survey design:**

- Design survey asking n_1 consumers about n_2 products.
- Can survey consumer i about product j only if they own it.
- Ask consumer i between c_i and c'_i questions.
- Ask between p_j and p'_j consumers about product j .

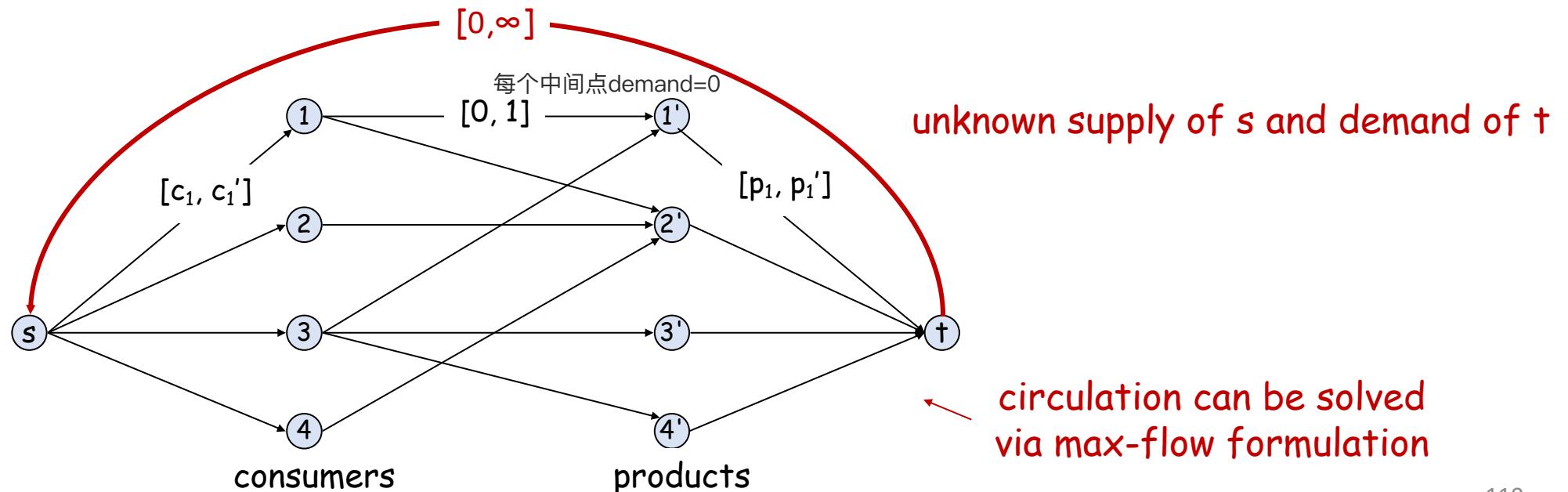
one question per
consumer-product

- **Goal.** Design a survey that meets these specs, if possible.

- **Bipartite perfect matching.** Special case when $c_i = c'_i = p_j = p'_j = 1$.

Survey Design: Circulation Formulation

- **Circulation formulation.** A circulation problem with **lower bounds**.
 - Add edge (i, j) if consumer i owns product j .
 - Add edge from s to consumer i . Add edge from product j to t .
 - Add edge from t to s with infinite capacity ∞ . **All demands = 0**.
- **Claim.** Integer circulation \Leftrightarrow feasible survey design.





南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

11. Airline Scheduling



Airline Scheduling

- **Airline scheduling:**
 - Complex computational problem faced by airline carriers.
 - Must produce schedules that are efficient in terms of equipment usage, crew allocation, and customer satisfaction.
 - One of largest consumers of high-powered algorithmic techniques.
 - **“Toy problem”:**
 - Manage flight crews by reusing them over multiple flights.
 - Input: set of k flights for a given day.
 - Flight i leaves origin o_i at time s_i and arrives at destination d_i at time f_i .
 - **Minimize** number of flight crews.
- even in presence of unpredictable events, e.g., weather and breakdowns
- one crew per flight

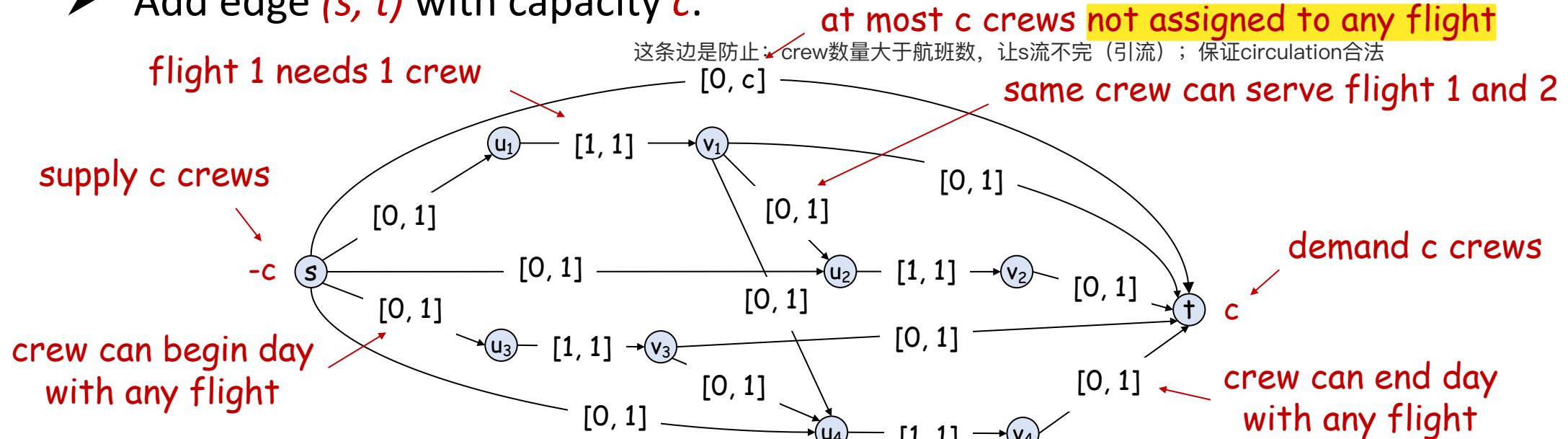


Airline Scheduling: Circulation Formulation

最多n组空乘, 1-n进行二分

- **Circulation formulation. (Check if c crews suffice.)**

- For flight i , add nodes u_i, v_i and edge (u_i, v_i) with lower bound and capacity 1.
- Add source s with demand $-c$, and edges (s, u_i) with capacity 1.
拆点保证了一定有下界
- Add sink t with demand c , and edges (v_i, t) with capacity 1. $u_i = \text{start of flight } i$
- If flight j reachable from i , add edge (v_i, u_j) with capacity 1. $v_i = \text{end of flight } i$
- Add edge (s, t) with capacity c .





Airline Scheduling: Running Time

- **Theorem.** Airline scheduling problem can be solved in $O(k^3 \log k)$ time.
- **Pf. (direct proof)**
 - k = number of flights
 - $O(k)$ nodes, $O(k^2)$ edges.
 - c = number of crews (unknown)
 - At most k crews needed \Rightarrow solve $\log_2 k$ circulation problems.
 - Any flow value is between 0 and $k \Rightarrow \leq k$ augmentations per circulation problem.
 - Overall time = $O(k^3 \log k)$.

binary search for min crew number c^*
最多k条增广路
- **Note.** Can solve in $O(k^3)$ time by formulating as minimum-flow problem.



Airline Scheduling: Running Time

- **Remark.** We solved a toy version of a real problem.
- **Real-world problem models countless other factors:**
 - Flight crews can fly only a certain number of hours in a given time window.
 - Need optimal schedule over planning horizon, not just one day.
 - Flights don't always leave or arrive on schedule.
 - Simultaneously optimize both flight schedule and fare structure.
- **Take-away message:**
 - Our solution is a generally useful technique for efficient reuse of limited resources but trivializes real airline scheduling problem.
 - Flow techniques are useful for solving airline scheduling problems (and are widely used in practice).
 - Running an airline efficiently is a very difficult problem.



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

12. Image Segmentation



Image Segmentation

- **Image segmentation.** Divide image into coherent regions.
 - Central problem in image processing.
- **Example.** Separate human and robot from background scene.

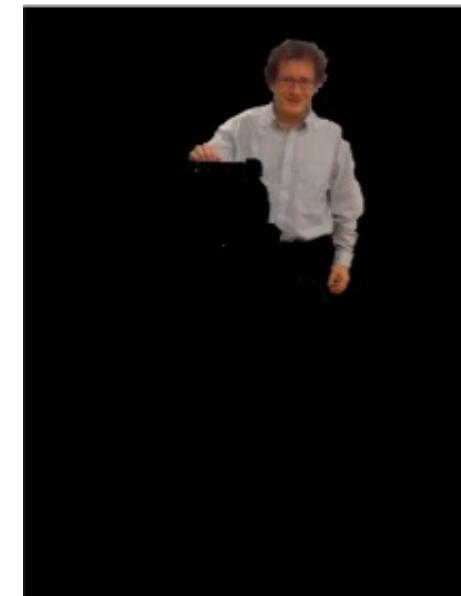
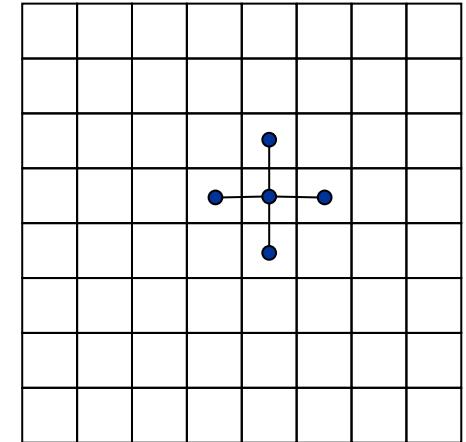




Image Segmentation

- **Foreground/background segmentation.** Label each pixel in picture as belonging to **foreground** or **background**.

- V = set of pixels, E = pairs of **neighboring** pixels.
- $a_i \geq 0$ is likelihood pixel i in foreground.
- $b_i \geq 0$ is likelihood pixel i in background.
- $p_{ij} \geq 0$ for each $(i, j) \in E$ is separation penalty for labeling one of i and j as foreground and the other as background.



- **Goals:**

- **Accuracy:** if $a_i > b_i$ in isolation, prefer to label i in foreground.
- **Smoothness:** if many neighbors of i are labeled foreground, we should be inclined to label i as foreground.
- Find **partition (A, B)** that maximizes $q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$

foreground background

相邻点且落在不同区域



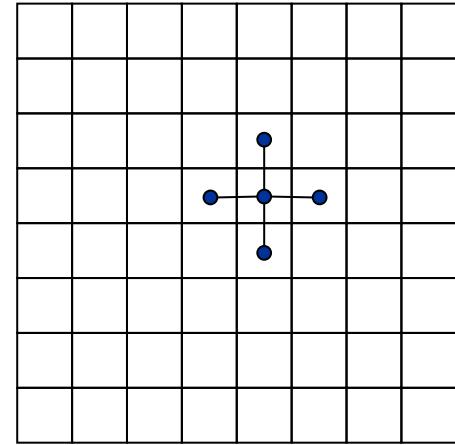
Image Segmentation

- **Formulate as min-cut problem:**

- No source or sink.
- Undirected graph.
- Maximization. ← how to convert max to min?

- **Turn into minimization problem:**

- Maximizing $q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$



is equivalent to minimizing $\left(\sum_{i \in A \cup B} a_i + \sum_{j \in A \cup B} b_j \right) - q(A, B) = \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}|=1}} p_{ij}$

↑
constant

Image Segmentation

- **Formulate as min-cut problem:**

- Include node for each pixel.
- Use **two anti-parallel edges** instead of undirected edge.
- Add **source s** to represent **foreground**.
- Add **sink t** to represent **background**.
- Denote the constructed flow network by $G' = (V', E', s, t, c)$.

edge in G :



antiparallel edges in G' :

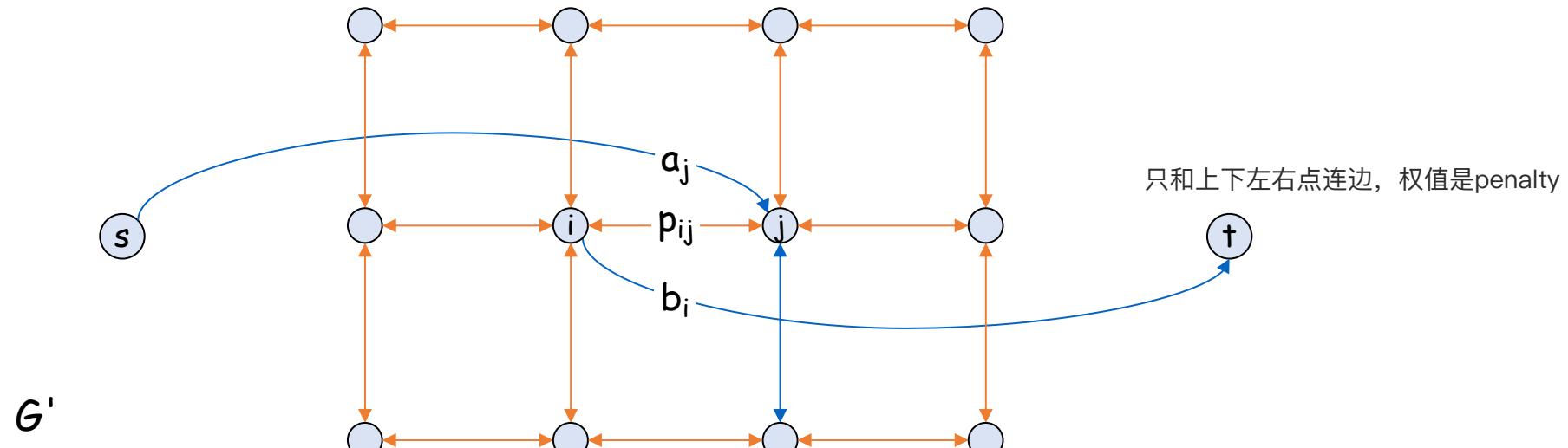
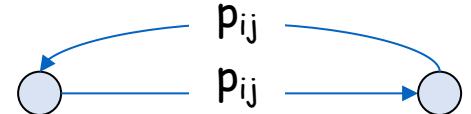
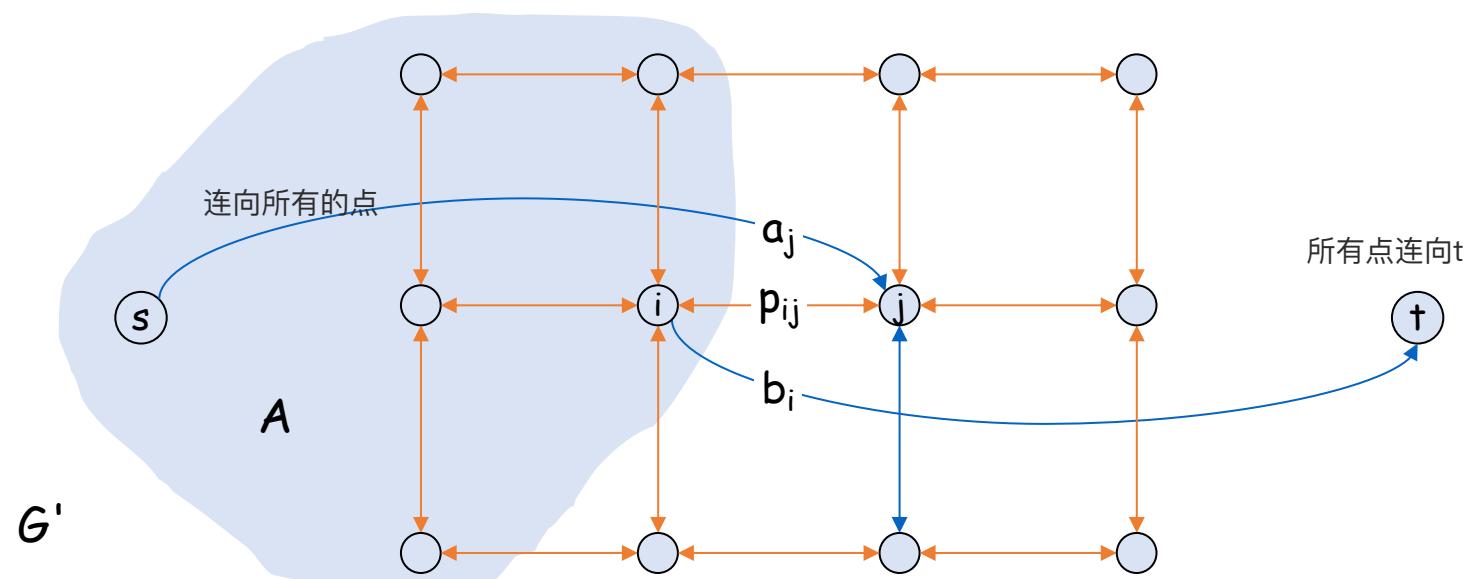




Image Segmentation

- **Min-Cut formulation.** Consider a cut (A, B) in directed graph G' :

- $A = \text{foreground}$ $c(A, B) = \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E \\ i \in A, j \in B}} p_{ij}$ ← if i and j on different sides,
 p_{ij} counted exactly once
 - Precisely the quantity we want to minimize in original undirected graph.





Network Flow: Closing Remarks

- **More applications:**

- Project selection (maximum weight closure problem). [section 7.11 of textbook]
- Baseball elimination. [section 7.12 of textbook]

- **Adding costs to max-flow problems:**

- Min-cost max flow (a natural extension to max-flow problems). **Lab 11**
- Min-cost perfect matching in bipartite graphs. **Assignment 5**