Applied Algorithm Design
Lecture 4

Pietro Michiardi

Institut Eurécom

Part 1: Network Flow - The fundamentals

Introduction (1)

In this Lecture we focus on a rich set of algorithmic problems that grow out of the first problems we formulated in Lecture 1: *Bipartite Matching*.

Reminder: Bipartite Graph

$G = (V, E)$ is an undirected graph whose node set can be partitioned as $V = X \cup Y$, with the property that every edge $e \in E$ has one end in X and the other end in Y.

Introduction (1)

Reminder: Matching

Collection of pairs over a set, with the property that no element of the set appears in more than one pair.

In a graph, the edges constitute pairs of nodes and we say that a matching in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M. M is a **perfect matching** if every node appears in exactly one edge of M.

Introduction (3)

Question: give a couple of applications of Matching in Bipartite Graphs

Introduction (4)

- One of the oldest problems in combinatorial algorithms is that of determining the size of the largest matching in a bipartite graph G.
- This problem turns out to be solvable by an algorithm that runs in polynomial time, but the development of this algorithm needs ideas very different from the techniques we've seen so far.
- Instead of developing the algorithm directly, we begin by formulating a very general class of problems: **Network Flow** problems, that include Bipartite Matching as a special case
- We develop a polynomial-time algorithm for the general problem of **Maximum-Flow Problem**

Flow Networks (1)

One often use graphs to model *Transportation networks*, that is networks whose edges carry some sort of traffic and whose nodes act as “switches” passing traffic between different edges.

Network models of this type have several ingredients:

- *capacities* on the edges, indicating how much traffic they can carry
- *source* nodes in the graph, which generate traffic
- *sink* nodes in the graph, which can “absorb” traffic as it arrives
- the traffic itself which is transmitted over the edges of the graph

Flow Networks (2)

We will refer to the traffic on these networks as **flow**, an abstract entity that is generated at source nodes, transmitted across edges, and absorbed at sink nodes.

Flow networks

A flow network is a *directed* graph $G = (V, E)$ with the following features:

- Associated with each edge e is a *capacity*, which is a non-negative number we denote c_e
- There is a single *source* node $s \in V$
- There is a single *sink* node $t \in V$

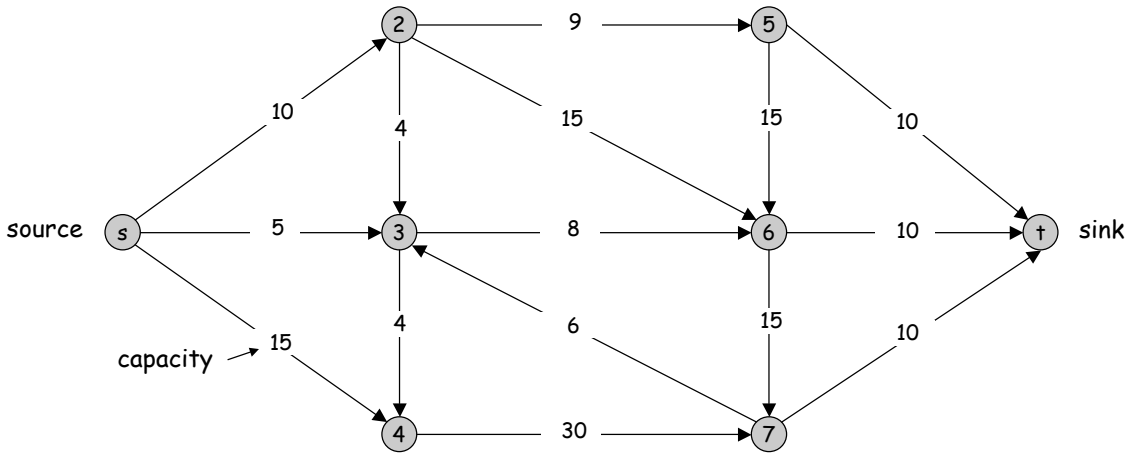
Nodes other than the source and the sink are called *internal* nodes.

Flow Networks: assumptions

We now enumerate the assumptions we'll make to simplify the analysis:

1. no edge enters the source s
2. no edge leaves the sink t
3. there is at least one edge incident to each node
4. all capacities are *integers*

And here's an example of a flow network:



Defining Flows (1)

Definition: Flows

We say that an *s-t flow* is a function f that maps each edge e to a non negative real number, $f : E \rightarrow \Re^+$

The value $f(e)$ intuitively represents the amount of flow carried by edge e. A flow must satisfy the following two properties:

Flow: Properties

- **Capacity conditions**: $\forall e \in E\,,\, 0 \leq f(e) \leq c_e$
- **Conservation conditions**: $\forall v \in V \backslash \{s, t\}$

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

Defining Flows (2)

In words, we have:

- The flow on an edge $f(e)$ cannot exceed the capacity of the edge
- For every node other than the source and the sink, the amount of flow entering must equal the amount of flow leaving
- The source generates flow
- The sink consumes flow

Defining Flows (2)

Definition: The value of a flow

The *value* of flow f, denoted $v(f)$ is defined to be the amount of flow generated at the source:

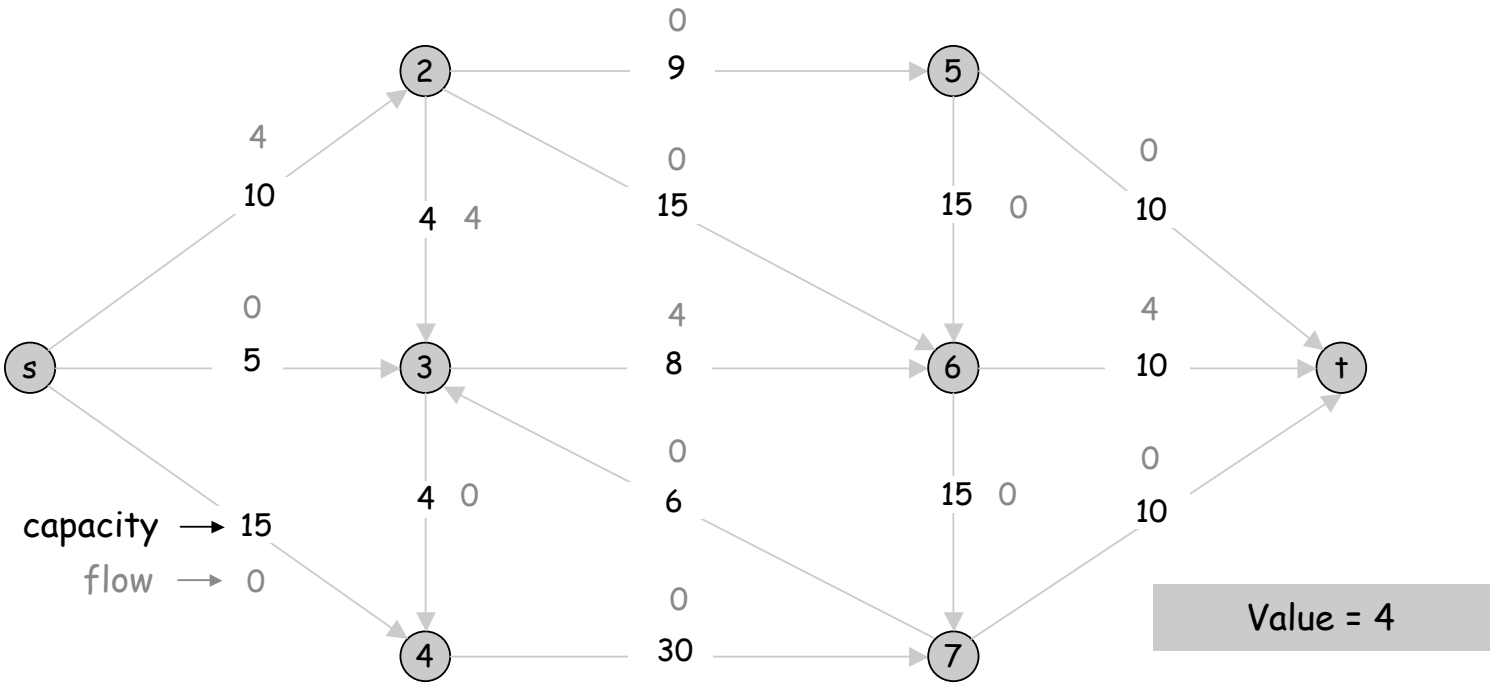
$$v(f) = \sum_{e \text{ out of } s} f(e)$$

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

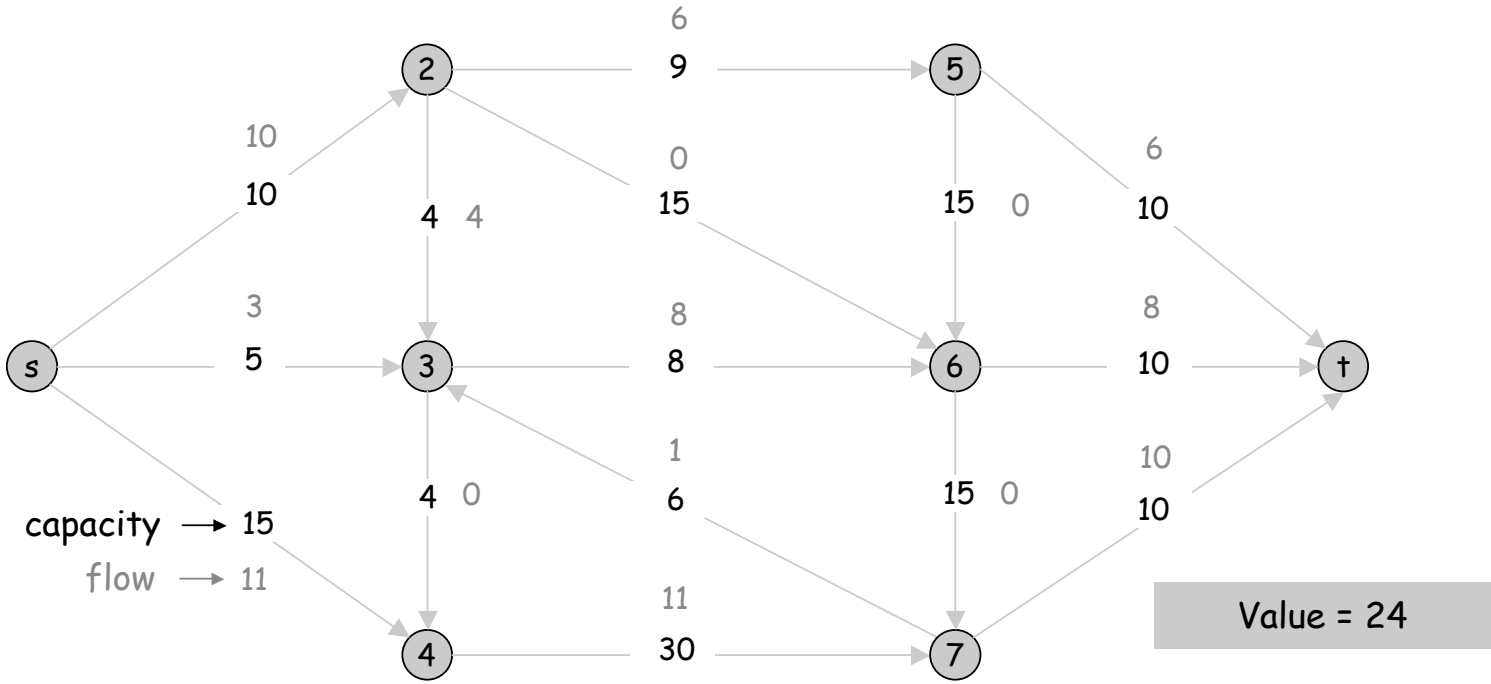
Let $S \subseteq V$:

$$f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e) \; ; \; f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$$

Defining Flows: examples (1)



Defining Flows: examples (2)



The Max-Flow Problem

Given a flow network, a natural goal is to arrange the traffic so as to make as efficient use as possible of the available capacity. Thus, the basic algorithmic problem we will consider is the following:

Definition: Max-Flow Problem

Given a flow network, find a flow of maximum possible value

Cuts in graphs (1)

- We will now introduce a useful construct that is somehow related to the problems we are addressing
- Given a graph $G = (V, E)$, we will find a way to partition its vertex set
- Remember we have seen already an example of a cut: bipartite graphs!

We will seek at finding the duality that exists between “cuts and flows” and exploit it to design an efficient algorithm for the max-flow problem. As a bonus we will obtain an algorithm for the dual problem of finding minimum capacity cuts.

Cuts in graphs (2)

Definition: $s - t$ cut

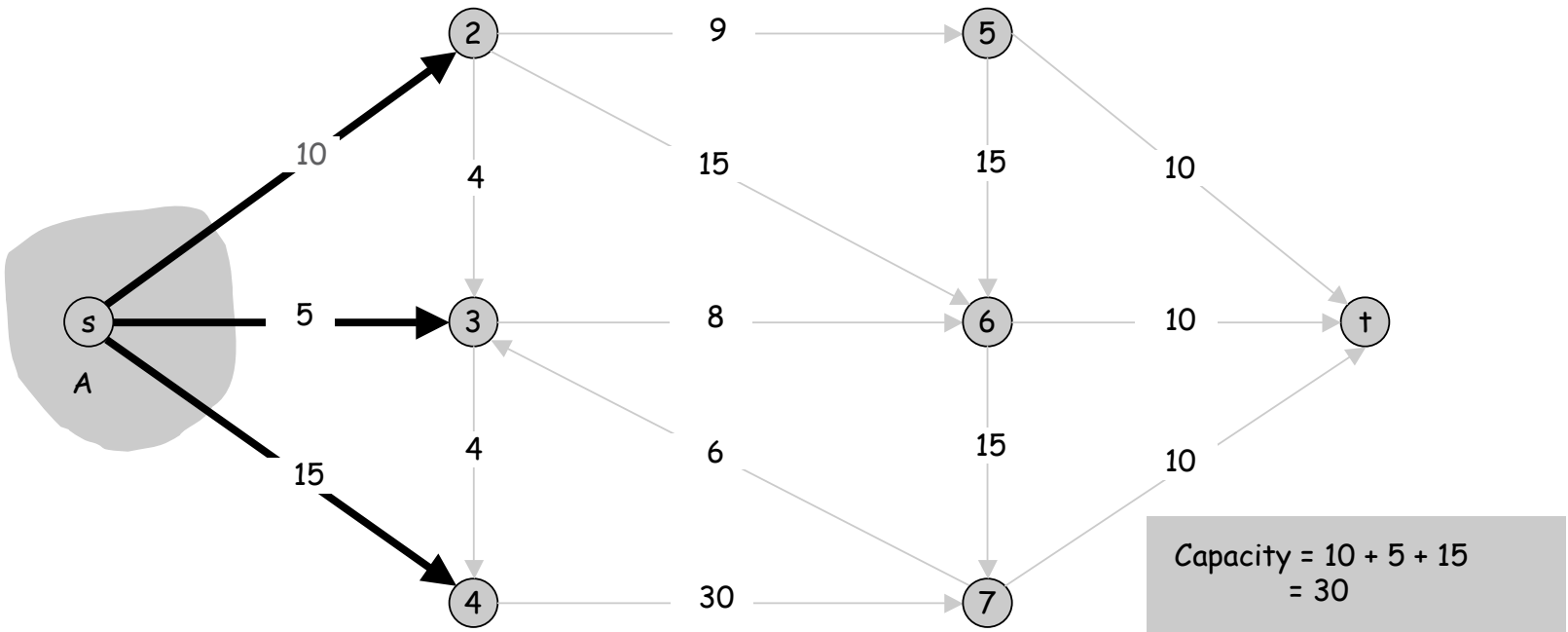
Given a directed graph $G(V, E)$ an $s - t$ *cut* is a partition (A, B) of V with $s \in A$ and $t \in B$

Definition: Capacity of a cut

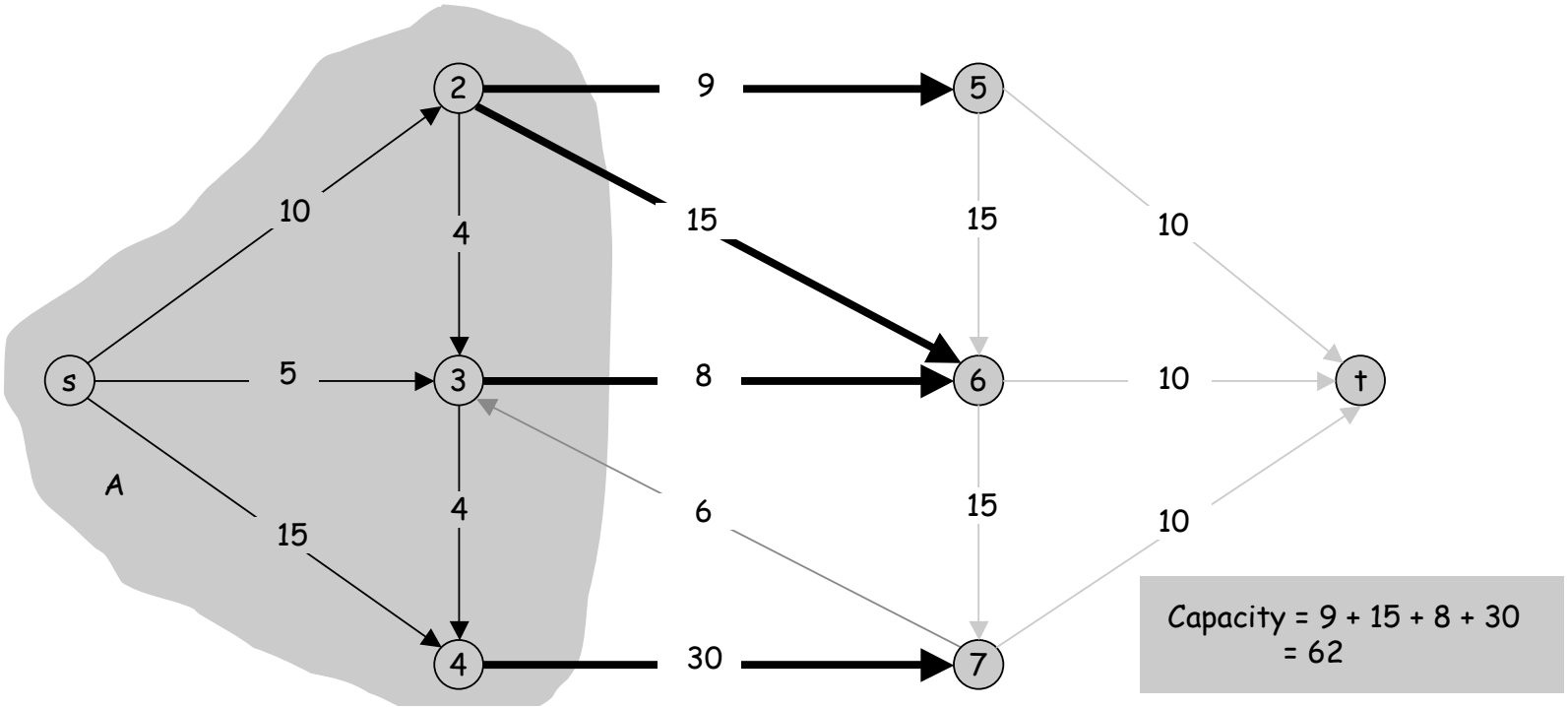
The *capacity* of a cut (A, B) is:

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

Cuts in graphs: example 1



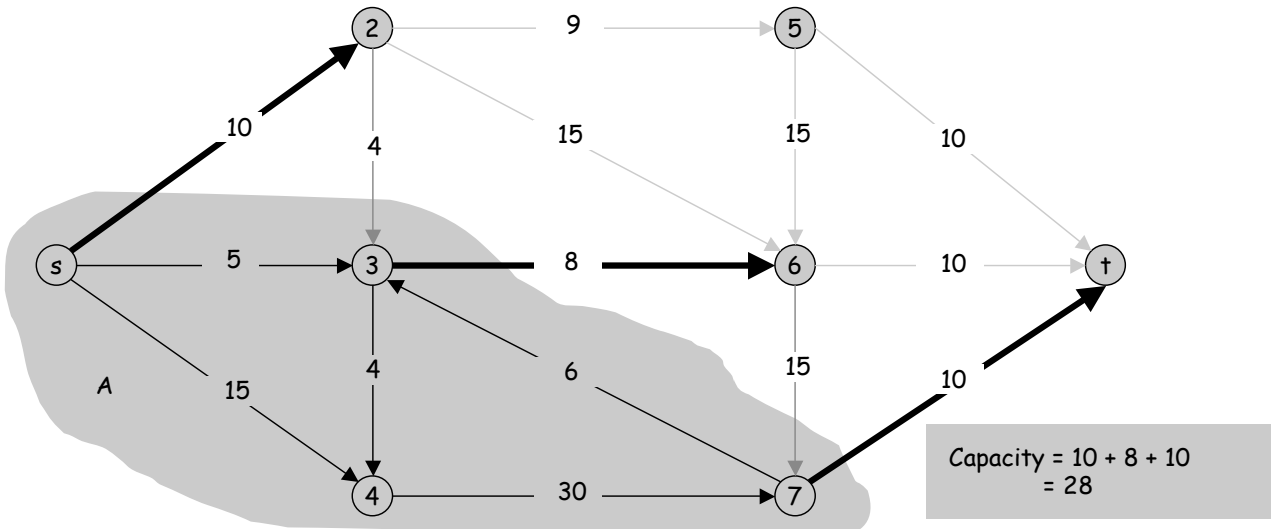
Cuts in graphs: example 2



The Min-Cut Problem

Definition: the Min-Cut Problem

The min-cut problem is to find an $s - t$ *cut* of minimum capacity.



Flows and cuts (1)

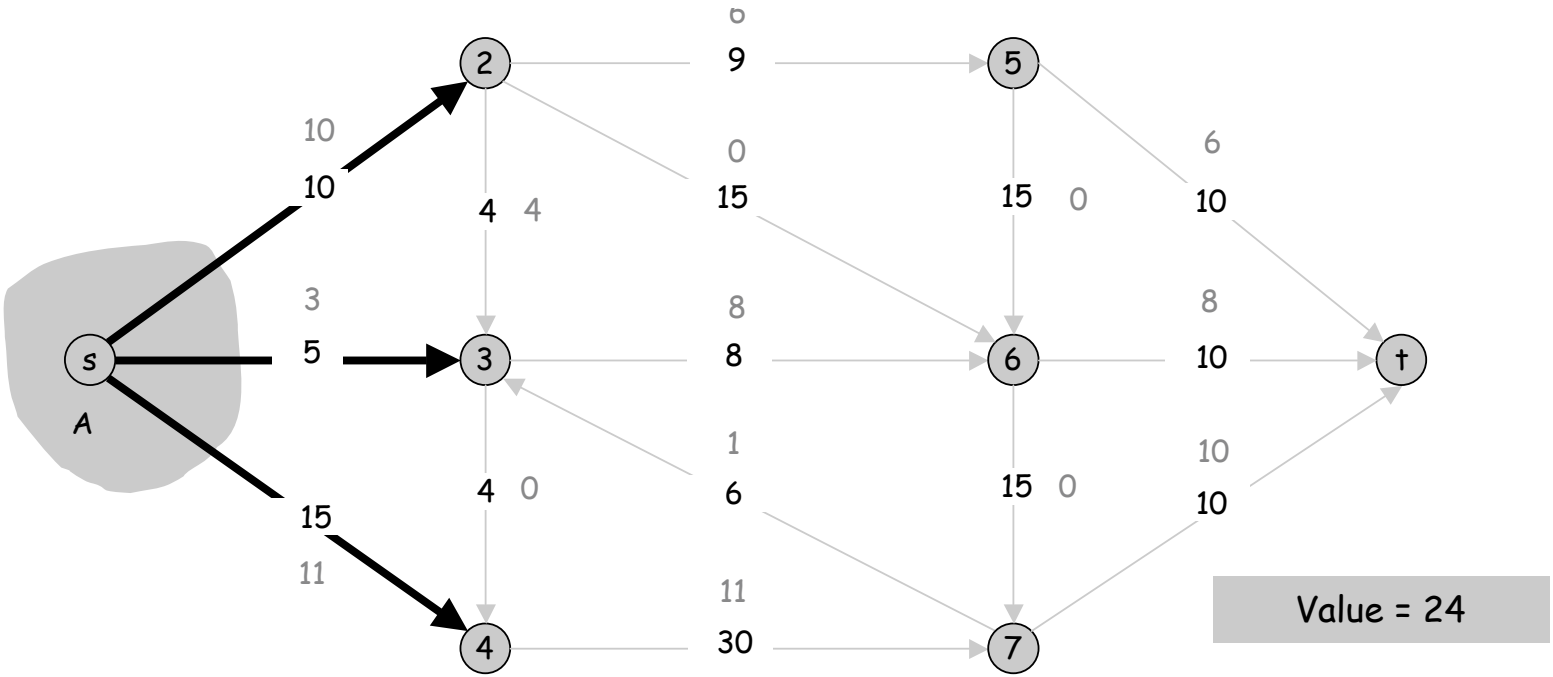
Lemma:

Let f be *any* flow, and let (A, B) be *any* $s - t$ cut.
Then, the net flow sent across the cut is equal to the amount leaving s.

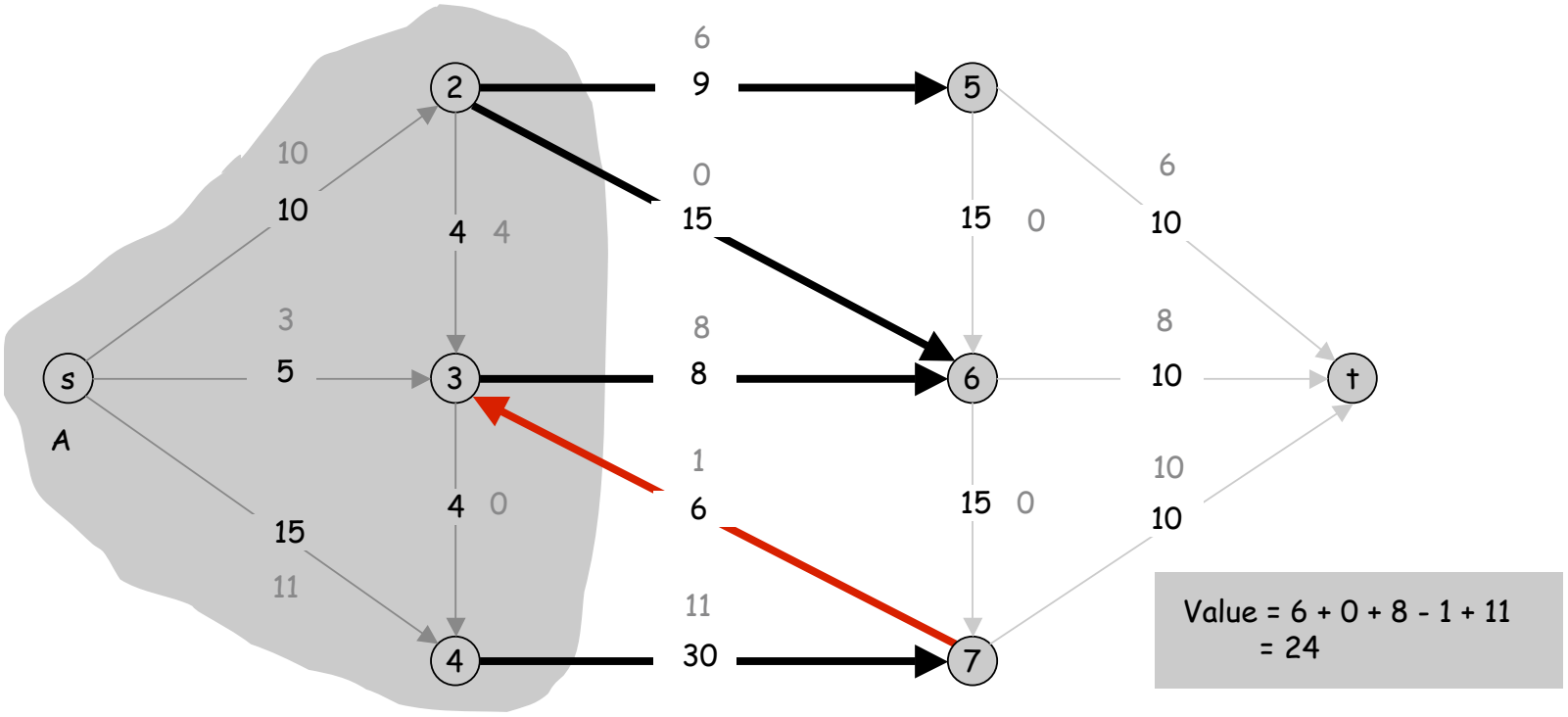
Consider dividing the nodes of the graph $G = (V, E)$ in two sets A and B so that $s \in A$ and $t \in B$. Any such division places an upper bound on the maximum possible flow value, since all the flow must cross from A to B somewhere.

Cuts turn out to provide very natural upper bounds on the values of flows, as expressed in the Lemma above. The Lemma is much stronger than a simple upper bound actually. It says that by watching the amount of flow f that goes across a cut, we can exactly *measure* the flow value: it is the total amount that leaves A minus the amount that "swirls" back into A.

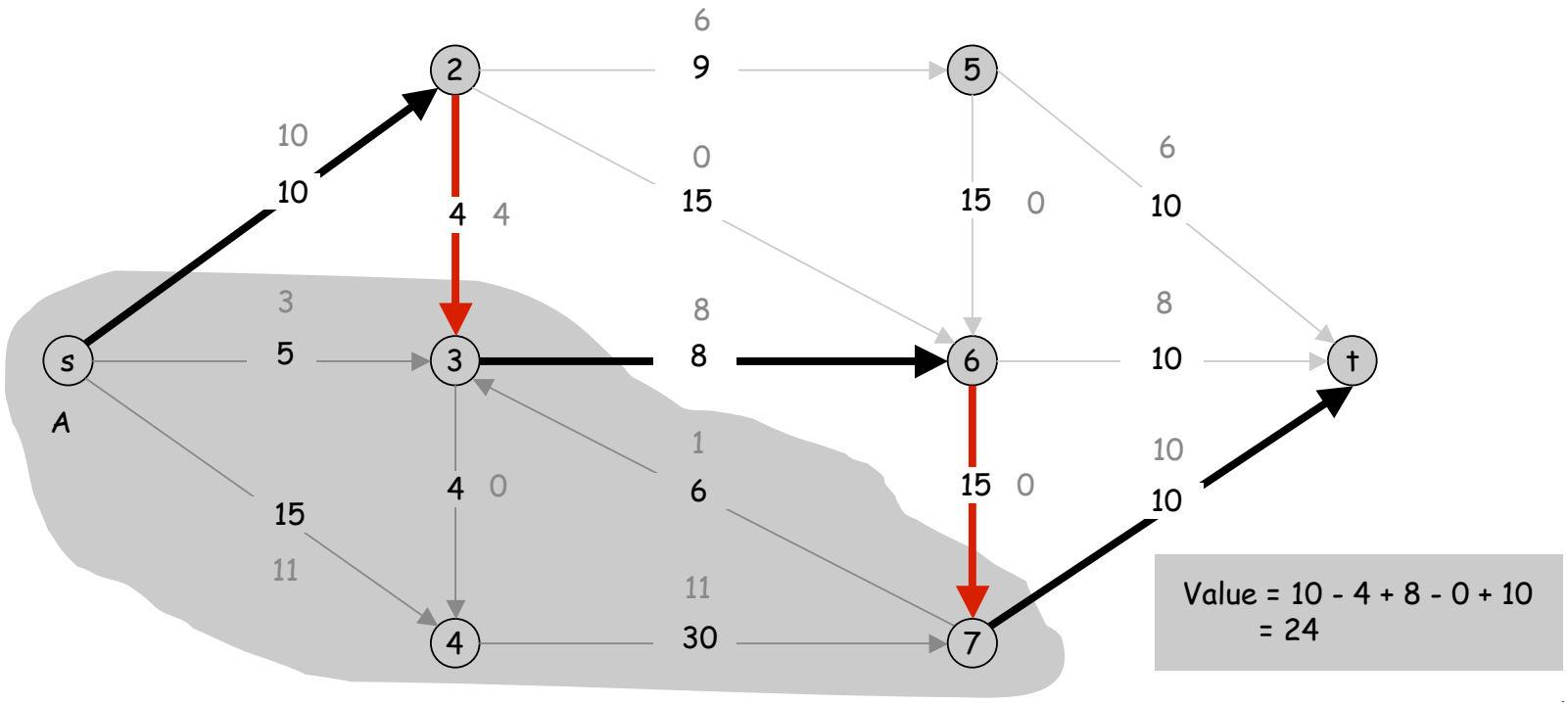
Flows and cuts: example 1



Flows and cuts: example 2



Flows and cuts: example 2



Flows and cuts (2)

Lemma:

Let f be *any* flow, and let (A, B) be *any* $s - t$ cut.
Then, the net flow sent across the cut is equal to the amount leaving s.

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

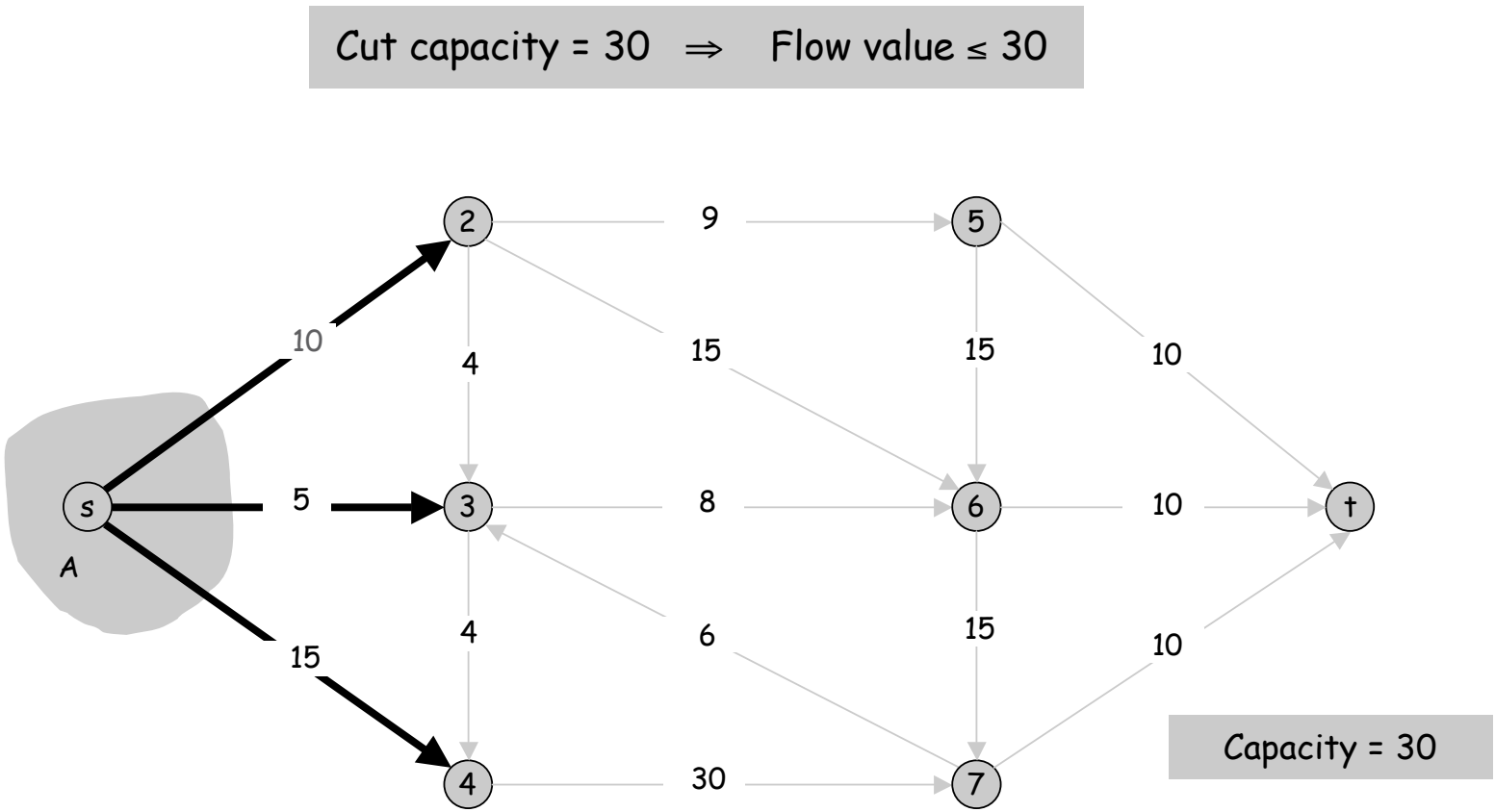
Proof.

$$\begin{aligned} v(f) &= \textstyle\sum_{e \text{ out of } s} f(e) \\ &= \textstyle\sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \right) \\ &= \textstyle\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \end{aligned}$$

Flows and cuts: weak duality (1)

Weak duality:

Let f be any flow and let (A, B) be any $s - t$ cut. Then the value of the flow is at most the capacity of the cut.



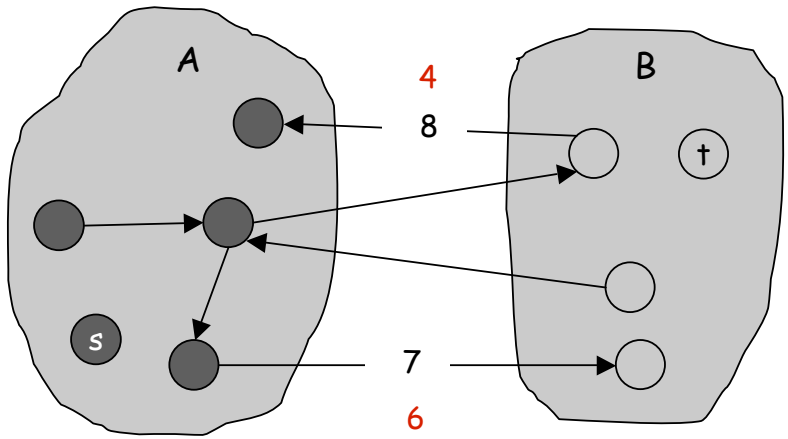
Flows and cuts: weak duality (1)

Weak duality:

Let f be any flow. Then for any $s - t$ cut we have:

$$v(f) \leq cap(A, B)$$

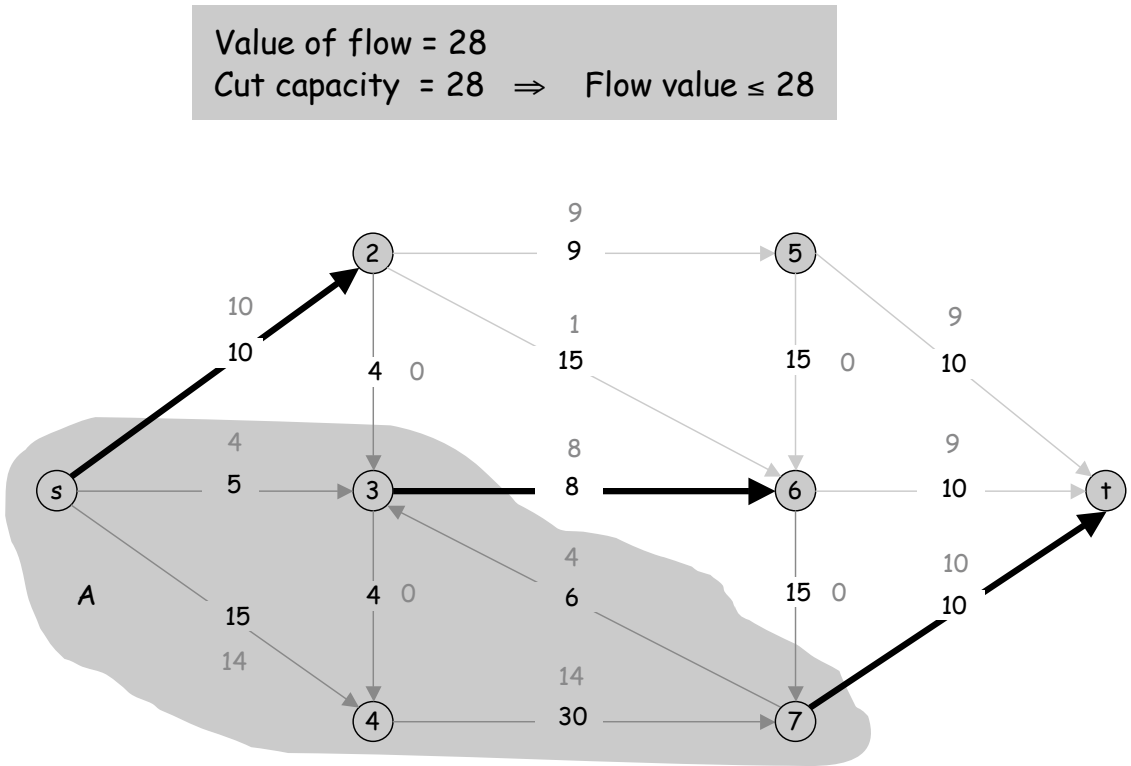
$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= \text{cap}(A, B) \quad \blacksquare \end{aligned}$$



Certificate of optimality

Corollary:

Let f be any flow, and let (A, B) be any cut.
If $v(f) = cap(A, B)$ then f is a max flow and (A, B) is a min cut.

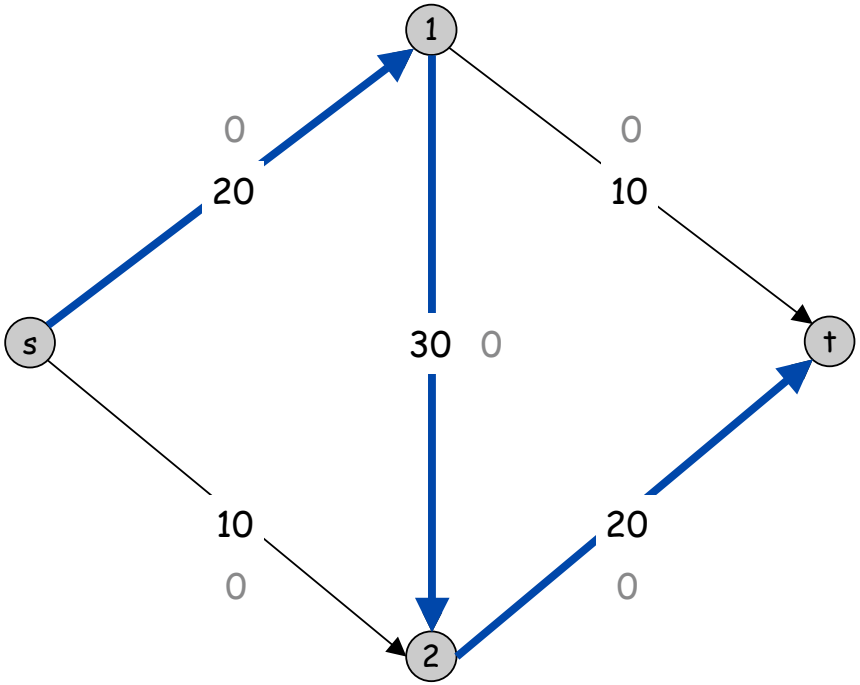


Towards a Max-Flow Algorithm

The Max-Flow problem turns out to be difficult to address with mostly of previous techniques we've seen in Lecture 3. Let's try with a simple, greedy approach.

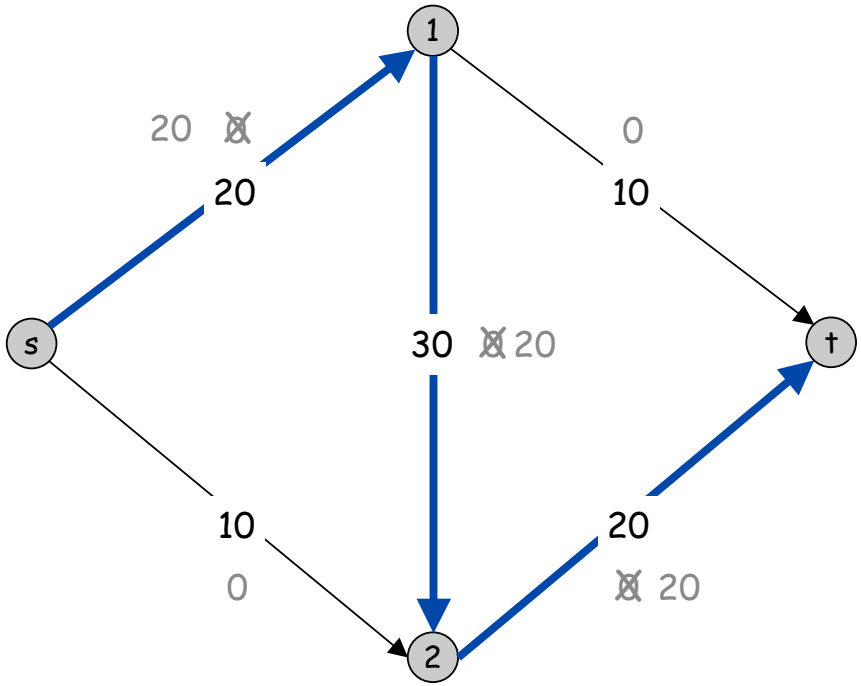
- Start with $f(e) = 0$ for all edge $e \in E$
- Find an $s - t$ path P where each edge has $f(e) < c(e)$
- "**Augment**" flow along path P
- Repeat until you get stuck

Towards a Max-Flow Algorithm: example (1)



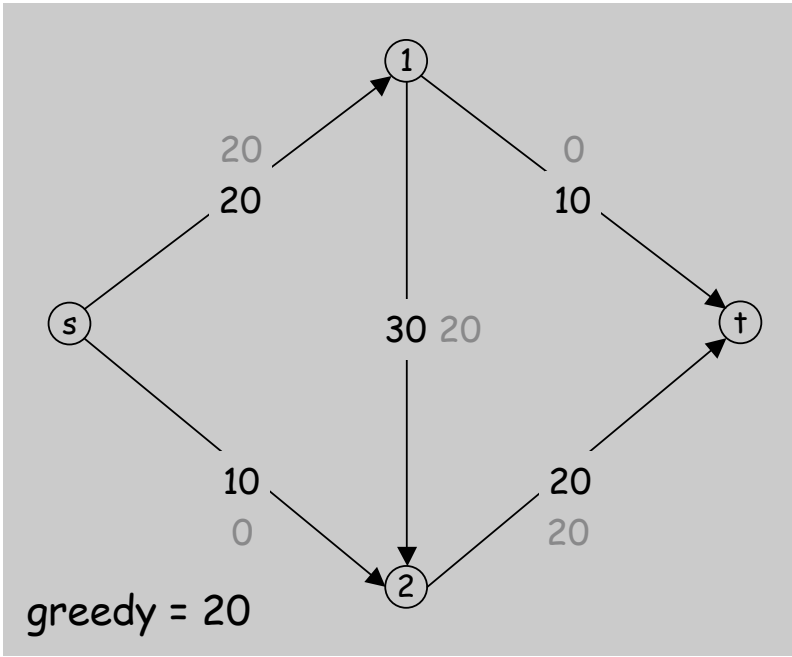
Flow value = 0

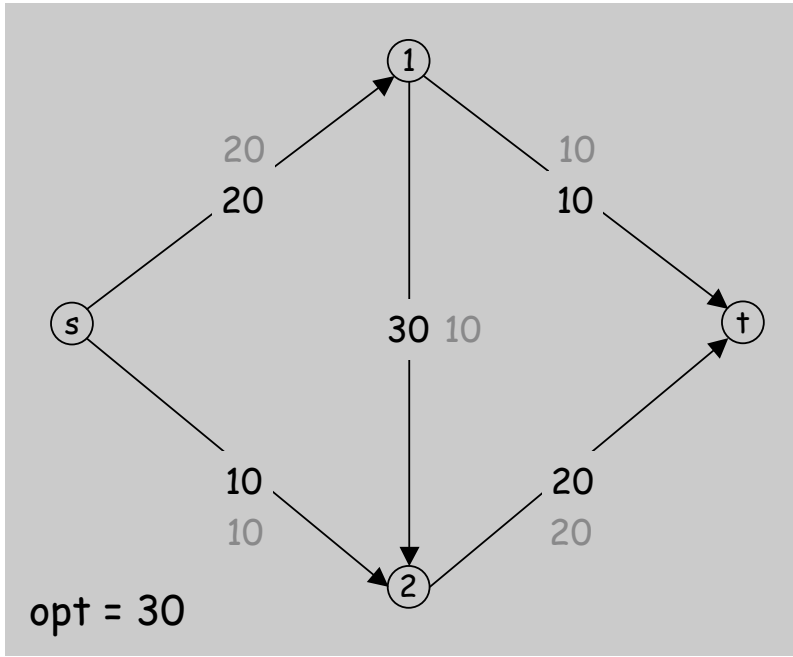
Towards a Max-Flow Algorithm: example (2)



Flow value = 20

Towards a Max-Flow Algorithm: example (3)





Towards a Max-Flow Algorithm: Observations

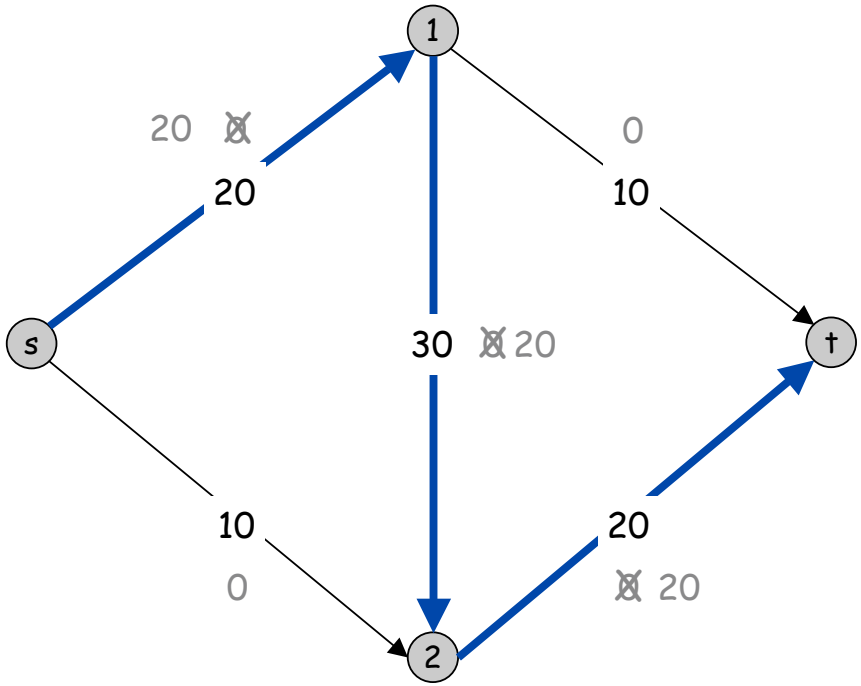
- What does it mean to augment the Flow, precisely?
- How can we make the algorithm outlined above more concrete?
- What is the difference between the optimal solution and the greedy?

Towards a Max-Flow Algorithm: Discussion (1)

Let's analyze informally what we've done so far.

- We started with zero flow: this clearly respects the capacity and conservation conditions
- We then try to increase the value of f by "pushing" flow along a path from s to t up to the limits imposed by edge capacities
- We select (for example) the path $< s, 1, 2, t >$ and increase the flow on each edge to 20, and leave $f(e) = 0$ on the other two edges

Towards a Max-Flow Algorithm: Discussion (2)



Flow value = 20

Towards a Max-Flow Algorithm: Discussion (3)

- In this way we still respect the capacity and conservation conditions
- The problem is that we are now stuck: there is no $s - t$ path on which we can *directly* push flow without exceeding some capacity and yet we do not have a maximum flow

We need a more general way of pushing flow from s to t so that in a situation like this we have a way to increase the value of the current flow.

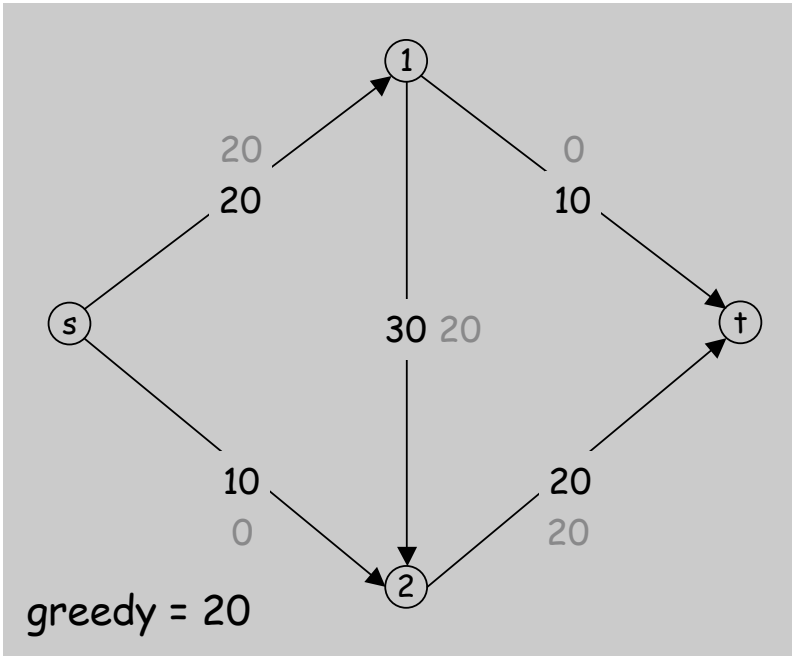
Towards a Max-Flow Algorithm: Discussion (4)

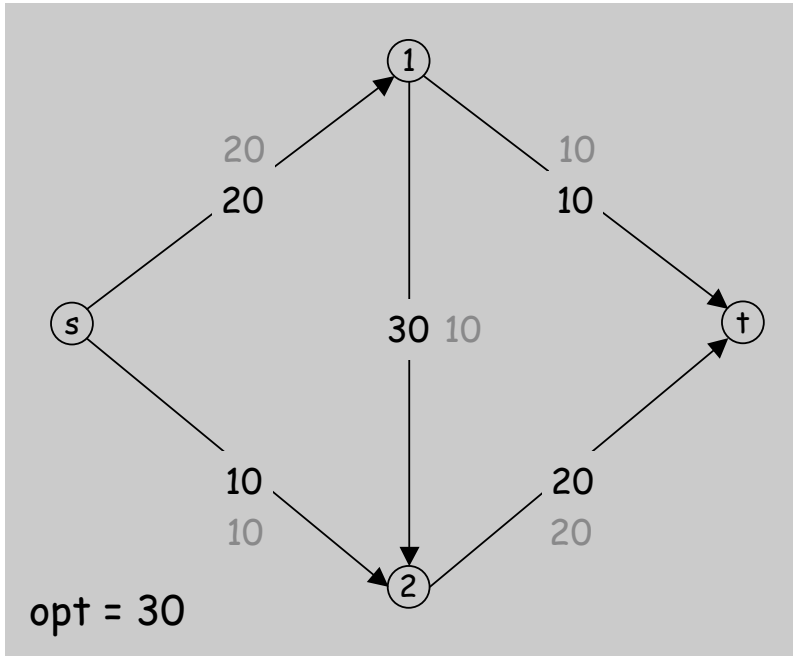
Intuition: towards Residual Graphs

Essentially, we'd like to:

- Push 10 units of flow along $< s, 2 > \rightarrow$ now there is too much flow coming into 2
- We "undo" 10 units of flow on $< 1, 2 > \rightarrow$ this restores conservation condition at 2, but results in too little flow leaving 1
- we push 10 units of flow along $< 1, t > \rightarrow$ we now have a valid flow, of maximum value

Towards a Max-Flow Algorithm: Discussion (5)





Residual Graph (1)

So we've described a more general way of pushing flow: we can push **forward** on edges that have leftover capacity, and we can push **backward** on edges that are already carrying flow.

Definition: Residual Graph

Given a flow network $G = (V, E)$, and a flow f on G, we define the residual graph G_f of G with respect to f as follows

- The node set of G_f is the same as that of G

Residual Graph (2)

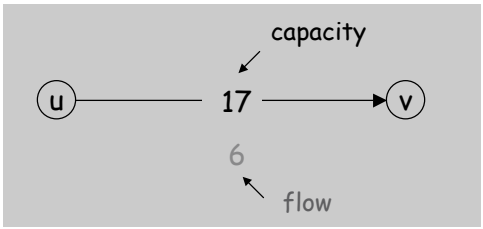
Definition: Residual Graph

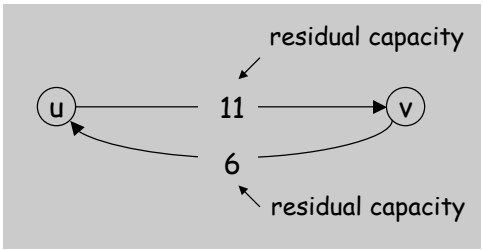
- For each edge $e = (u, v) \in G$ on which $f(e) < c_e$ there are $c_e - f(e)$ residual units of capacity on which we could try pushing forward.
  $\rightarrow$ we include the edge $e = (u, v)$ in G_f with a capacity of $c_e - f(e)$ and call this a **forward** edge

- For each edge $e = (u, v) \in G$ on which $f(e) > 0$, there are $f(e)$ units of flow that we can undo if we want to, by pushing backward.
  $\rightarrow$ we include the edge $e' = (v, u)$ in G_f, with a capacity of $f(e)$ and call this a **backward** edge

Residual Graph: example

- **Original Edge**:
  $e = (u, v) \in E$.
  - Flow $f(e)$, capacity $c(e)$
- **Residual Edge**:
  - Undo flow sent
  - $e = (u, v)$ and $e^R = (u, v)$
  - Residual capacity:





$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

- **Residual Graph**: $G_f = (V, E_f)$
  - Residual edges with positive residual capacity
  - $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$

Augmenting paths in Residual Graphs (1)

We now want to make precise the way we push flow from s to t in G_f.

- Let P be a simple $s - t$ path in G_f, that is a path that does not visit any node more than once

Definition: bottleneck

We define `Bottleneck`(P,f) to be the **minimum** residual capacity of any edge on P, with respect to the flow f

Augmenting paths in Residual Graphs (2)

Algorithm 1: augment(P,f)

Let $b = \texttt{Bottleneck}(P,f)$
foreach $e = (u, v) \in P$ **do**
    if $e = (u, v)$ *is a forward edge* **then**
        increase $f(e) \in G$ by b
    else
        $e = (u, v)$ is a backward edge
        $e = (v, u)$
        decrease $f(e) \in G$ by b
    end
end

The Ford-Fulkerson Algorithm

Algorithm 2: FF(G, s, t, c)

foreach $e \in E$ **do**
  $f(e) \leftarrow 0$
end
$G_f \leftarrow$ residual graph
while *(there exists augmenting path P in G_f)* **do**
  Let P be a simple $s - t$ path in G_f
  $f' \leftarrow$ Augment(P,f)
  $f \leftarrow f'$
  update G_f to $G_{f'}$
end
return f

The Ford-Fulkerson Algorithm: Demo

Believe it or not! A demo!!

Max-Flows and Min-Cuts (1)

Let $\bar{f}$ denote the flow that is returned by the Ford-Fulerkson algorithm. We want to show that $\bar{f}$ has the maximum possible value of any flow in G and we do this as follows:

- we exhibit an $s - t$ cut (A^*, B^*) for which $v(\hat{f}) = c(A^*, B^*)$
- this immediately establishes that $\hat{f}$ has the maximum value of any flow
- it also tells us that (A^*, B^*) has the minimum capacity of any $s - t$ cut

Max-Flows and Min-Cuts(2)

The Ford-Fulerkson algorithm terminates when the flow f has no $s-t$ path left in the residual graph G_f. This turns out to be the only property needed for proving its optimality.

Theorem: augmenting path theorem

If f is an $s-t$ flow such that there is no $s-t$ path in the residual graph G_f, then there is an $s-t$ cut (A^*, B^*) in G for which $v(\hat{f}) = c(A^*, B^*)$. Consequently, f has the maximum value of any flow in G and (A^*, B^*) has the minimum capacity of any $s-t$ cut in G.

Theorem: Max-Flow Min-Cut theorem

The value of the max flow is equal to the value of the min cut.

We sketch the proofs of these theorems next.

Max-Flows and Min-Cuts(3)

We prove both theorems simultaneously showing that the following are equivalent:

1. There exists a cut (A, B) such that $v(f) = c(A, B)$
2. Flow f is a max flow
3. There is no augmenting path relative to f

Max-Flows and Min-Cuts(4)

Proof.

- $1 \Rightarrow 2$: This was the corollary to weak duality lemma
- $2 \Rightarrow 3$: We show contra-positive. Let f be a flow. If there exists an augmenting path, then we can improve f by sending flow along that path
- $3 \Rightarrow 1$:
  - Let f be a flow with no augmenting paths
  - Let A be set of vertexes reachable from s in residual graph
  - By definition of A, $s \in A$
  - By definition of f, $t \notin A$

Running Time of the Ford-Fulkerson algorithm

- **Assumption**: all capacities are integers $\in [1, C]$
- **Invariant**: Every flow value $f(e)$ and every residual capacities $c_f(e)$ remains an integer throughout the algorithm

Theorem:

The F-F algorithm terminates in at most $v(f^*) \leq nC$ iterations

Proof.

Each augmentation increases value of the flow by at least 1 □

Running Time of the Ford-Fulkerson algorithm

Corollary:

If $C = 1$, the F-F algorithm runs in $O(mn)$ time

Integrity Theorem:

If all capacities are integers, then there exists a max flow f for which every flow value $f(e)$ is an integer

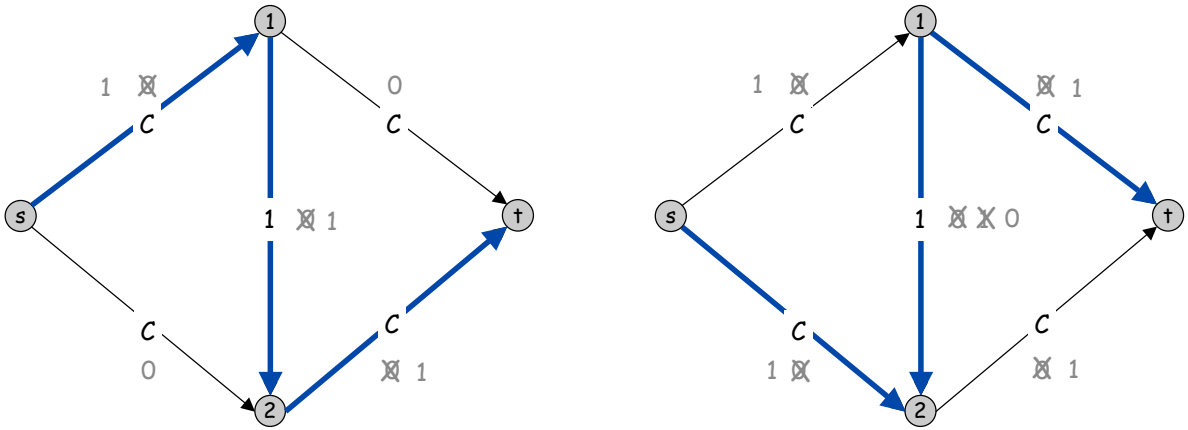
Proof.

Since algorithm terminates, theorem follows from invariant. □

Choosing good augmenting paths (1)

The execution of the F-F algorithm can end up in an exponential number of augmentations!

- Question: is the generic F-F algorithm polynomial in input size?
- Answer: No. If max capacity is C, the the algorithm can take C iterations.



Choosing good augmenting paths (2)

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms
- Clever choices lead to polynomial algorithms
- If capacities are irrational, algorithm not guaranteed to terminate

Goal: choose augmenting paths so that:

- You can find augmenting paths efficiently
- There are few iterations

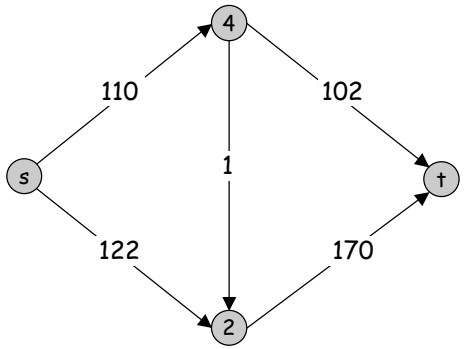
Chose augmenting paths with: [Edmonds-Karp 1972]

- Max bottleneck capacity
- Sufficiently large bottleneck capacity
- Fewest number of edges

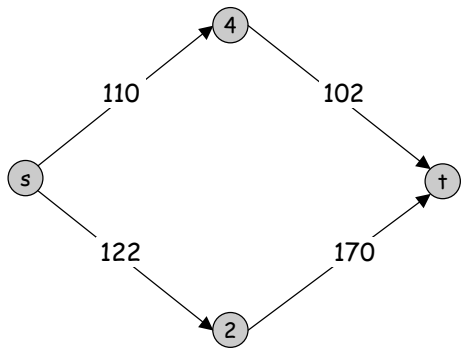
Capacity scaling (1)

The intuition here is that choosing paths with highest bottleneck capacity increases flow by max possible amount.

- Don't worry about finding exact highest bottleneck path
- Maintain a scaling parameter Δ
- Let $G_f(\Delta)$ be the sub-graph of the residual graph consisting of only arcs with capacity at least Δ



G_f



$G_f(100)$

Capacity scaling (2)

Algorithm 3: Scaling-Max-Flow(G, s, t, c)

foreach $e \in E$ **do**
  $f(e) \leftarrow 0$
end
$\Delta \leftarrow$ smallest power of $2 \geq C$
$G_f \leftarrow$ residual graph
while $(\Delta \geq 1)$ **do**
  $G_f(\Delta) \leftarrow \Delta-$residual graph
  while ($\exists$ *augmenting path* $P \in G_f(\Delta)$) **do**
    $f \leftarrow$ augment(f, c, P)
    update $G_f(\Delta)$
  end
  $\Delta \leftarrow \Delta/2$
end
return f

Capacity scaling: correctness

- **Assumption**: all edge capacities are integers $\in [1, C]$
- **Integrity invariant**: all flow and residual capacity values are integer

Theorem: Correctness

If the algorithm terminates, then f is a max flow

Proof.

- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths

□

Capacity scaling: running time

Lemma 1:

The outer while loop repeats $1 + \lceil log_2 C \rceil$ times

Proof.

Initially $C \leq \Delta < 2C$. Δ decreases by a factor of 2 at each iteration $\square$

Lemma 2:

Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow is at most $v(f) + m\Delta$

Capacity scaling: running time

Lemma 3:

There are at most $2m$ augmentations per scaling phase.

- Let f be the flow at the end of the previous scaling phase
- Lemma 2 $\Rightarrow v(f^*) \leq v(f) + m(2\Delta)$
- Each augmentation in a Δ-phase increases $v(f)$ by at least Δ

Theorem:

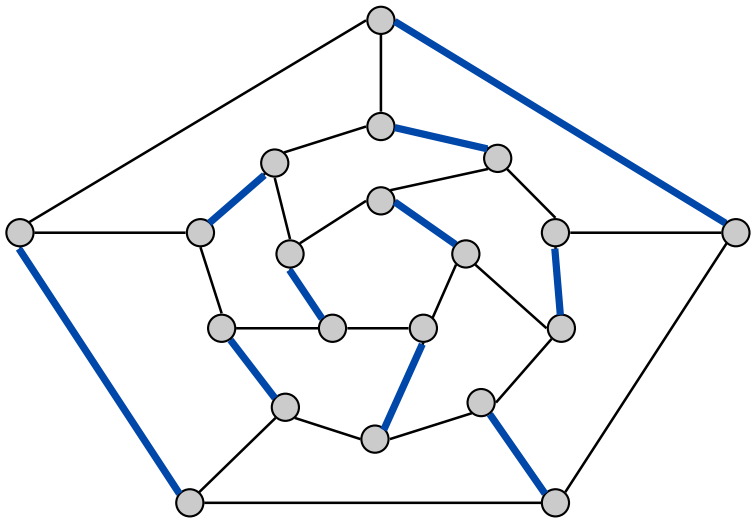
The scaling max-flow algorithm finds a max flow in $O(m^2 logC)$ augmentations. It can be implemented to run in $O(m^2 logC)$ time.

Part 2: Network Flow - Applications

Matching

Definition: Matching

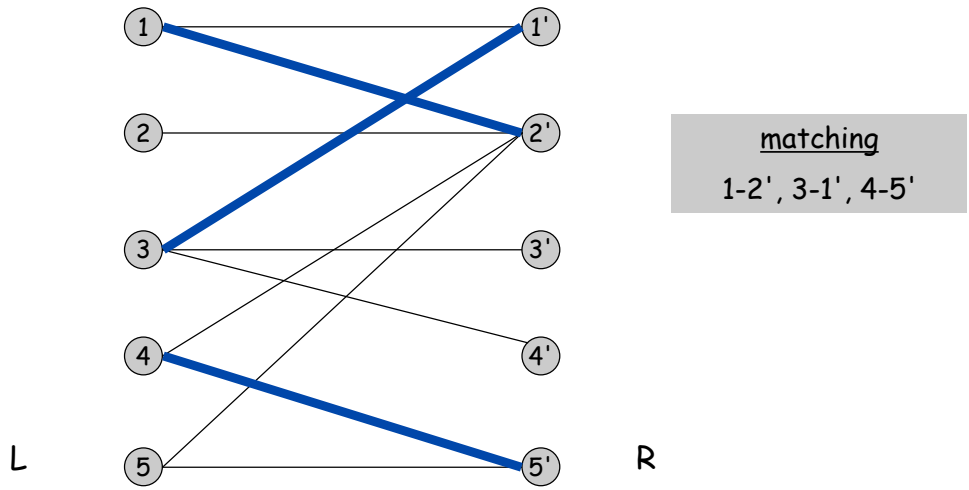
- Input: undirected graph $G = (V, E)$
- $M \subseteq E$ is a matching if each node appears in at most one edge in M
- Max Matching: find a max cardinality matching



Bipartite Matching (1)

Definition: Bipartite Matching

- Input: undirected, **bipartite** graph $G = (V, E)$
- $M \subseteq E$ is a matching if each node appears in at most one edge in M
- Max Matching: find a max cardinality matching

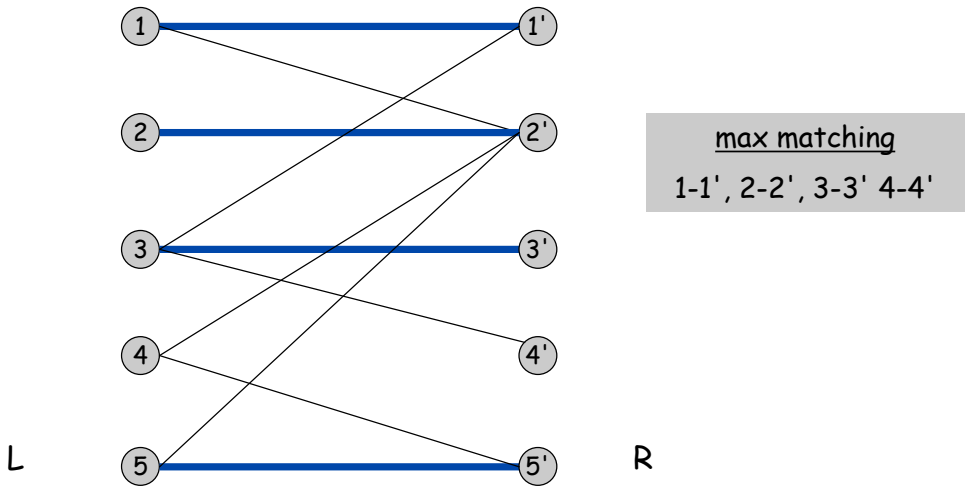


4

Bipartite Matching (2)

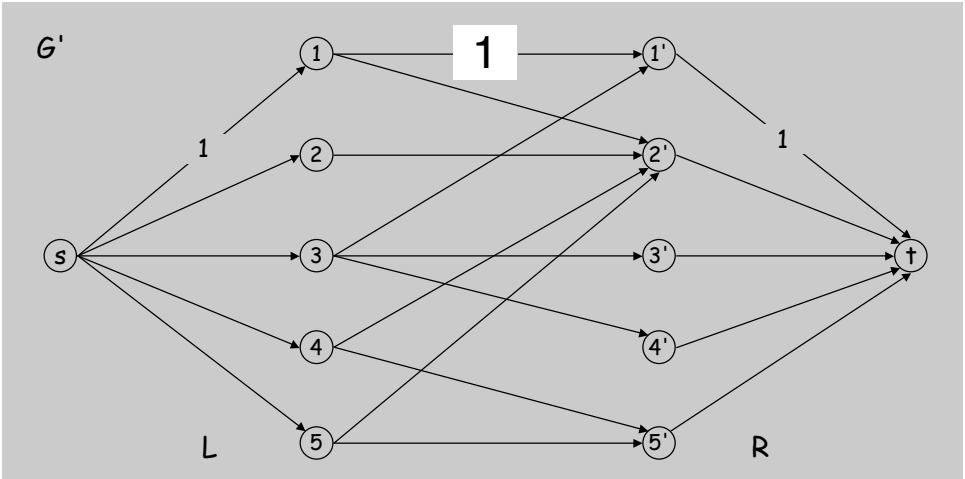
Definition: Bipartite Matching

- Input: undirected, **bipartite** graph $G = (V, E)$
- $M \subseteq E$ is a matching if each node appears in at most one edge in M
- Max Matching: find a max cardinality matching



Bipartite Matching: Max-Flow Formulation

- Create a digraph $G' = (L \cup R \cup \{s, t\}, E')$
- Direct all edges from L to R and assign **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



Bipartite Matching: Proof of correctness (1)

Theorem:

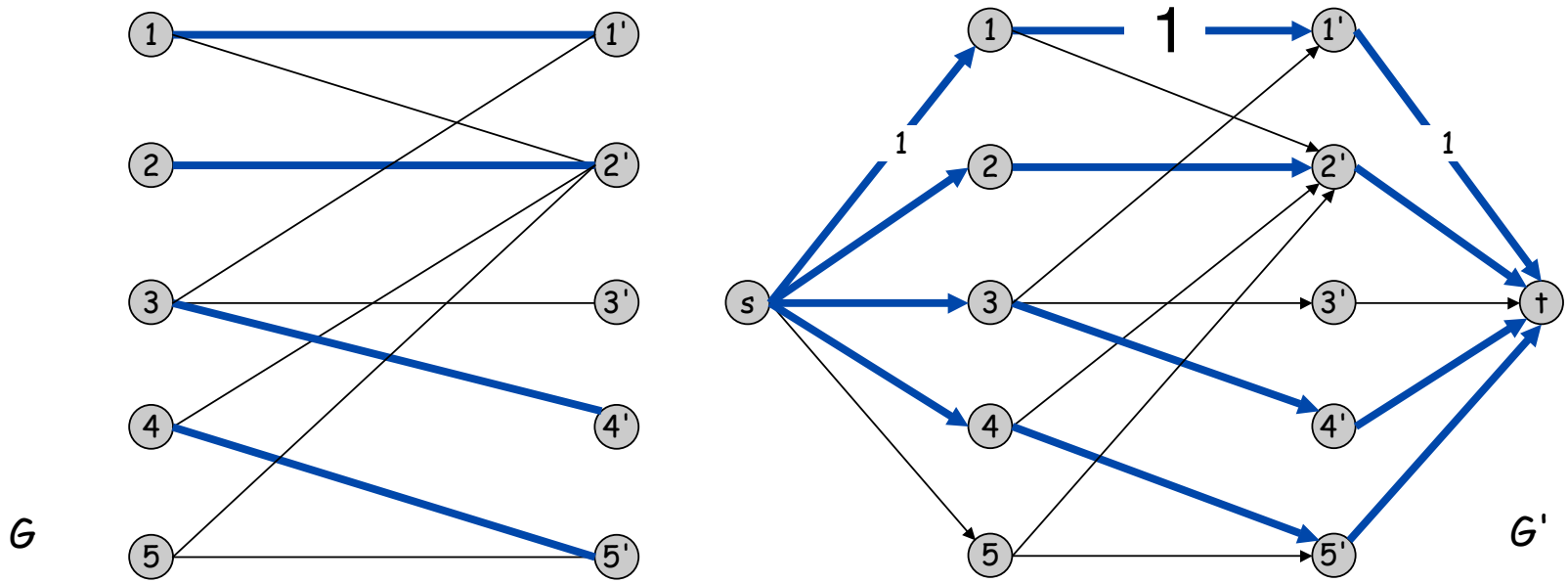
Max cardinality matching in G is equal to the value of max flow in G'

Proof.

We now prove $\leq$

- Given max matching M of cardinality k
- Consider flow f that sends 1 unit along each of k paths
- f is a flow, and has cardinality k

Bipartite Matching: Proof of correctness (2)



Bipartite Matching: Proof of correctness (3)

Theorem:

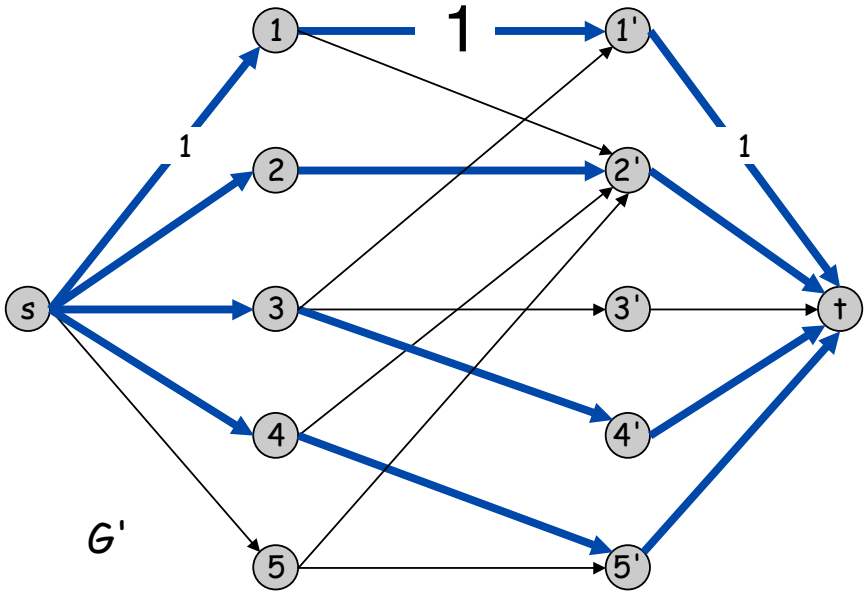
Max cardinality matching in G is equal to the value of max flow in G'

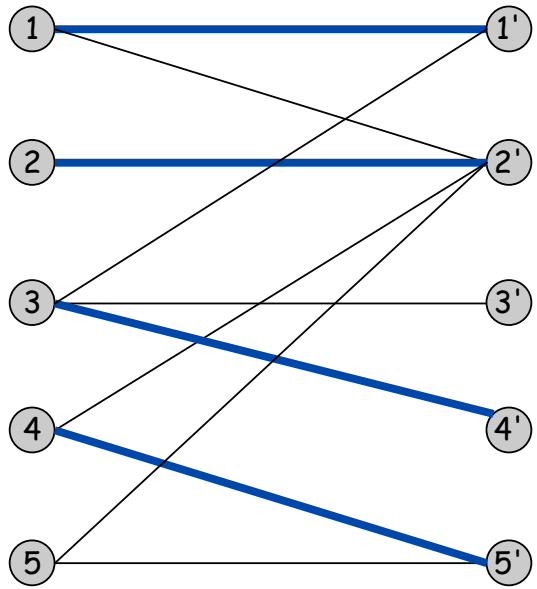
Proof.

We now prove $\geq$

- Let f be a max flow in G' of value k
- Integrality theorem $\Rightarrow$ k is integral hence we can assume f takes $0-1$ value
- Consider M = set of edges from L to R with $f(e) = 1$
  - Each node in L and R participates in at most one edge in M
  - $|M| = k$: consider cut $(L \cup s, R \cup t)$

Bipartite Matching: Proof of correctness (4)





Perfect Matching

Definition:

A matching $M \subseteq E$ is **perfect** if each node appears in exactly one edge in M

Question: When does a bipartite graph have a perfect matching?

Structure of bipartite graphs with perfect matching:

- We must have $|L| = |R|$
- What other conditions are necessary?
- What conditions are sufficient?

Perfect Matching (1)

Notation:

Let S be a subset of nodes, and let $N(S)$ be the set of nodes adjacent to node in S

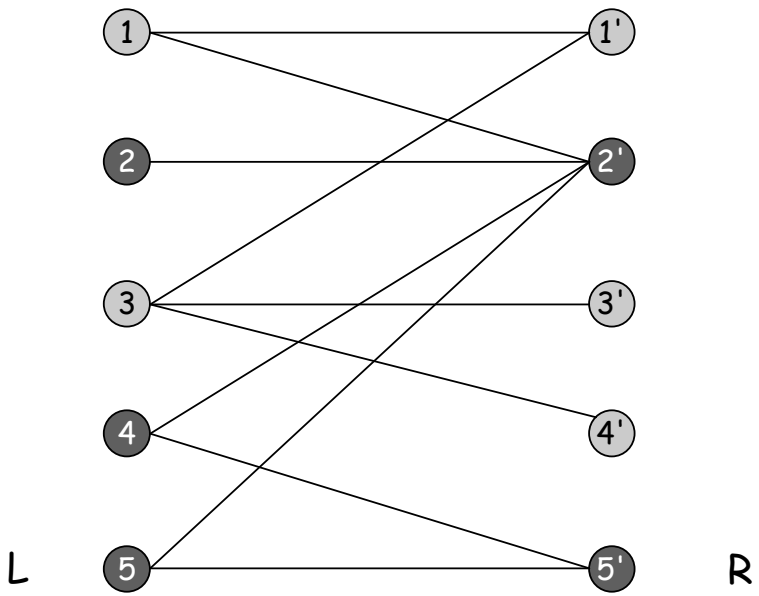
Observation:

If a bipartite graph $G = (L \cup R, E)$ has a perfect matching then $|N(S)| \geq |S|$ for all subsets $S \subseteq L$

Proof.

Each node in S has to be matched to different node in $N(S)$ □

Perfect Matching (2)



No perfect matching:

S = { 2, 4, 5 }

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

Bipartite Matching: running time

Question: remember what was the running time of the Gale-Shapley algorithm we saw in Lecture 1?

Which max flow algorithm to use for bipartite matching?

- Generic augmenting path: $O(m\,v(f^*)) = O(mn)$
- Capacity scaling: $O(m^2 log C) = O(m^2)$
- Shortest augmenting path[a]: $O(m\sqrt{n})$

[a] Not seen here.

Non-Bipartite Matching

Non-bipartite matching:

- Structure of non-bipartite graphs is more complicated, but well understood: see [Tutte-Berge, Edmonds-Galai]
- Blossom algorithm: $O(n^4)$ [Edmonds 1965]
- Best known: $O(m\sqrt{n})$ [Micali-Vazirani 1980]

Disjoint Paths: introduction

In the first part, we described a flow f as a kind of "traffic" in the network. However, the actual definition we used has a static twist: for each edge e, we simply specify a number $f(e)$ saying the amount of flow crossing e.

Let's see if we can revive the more dynamic, traffic-oriented picture a bit and try formalizing the sense in which units of flow "travel" from the source to the sink.

Edge Disjoint Paths: the Problem

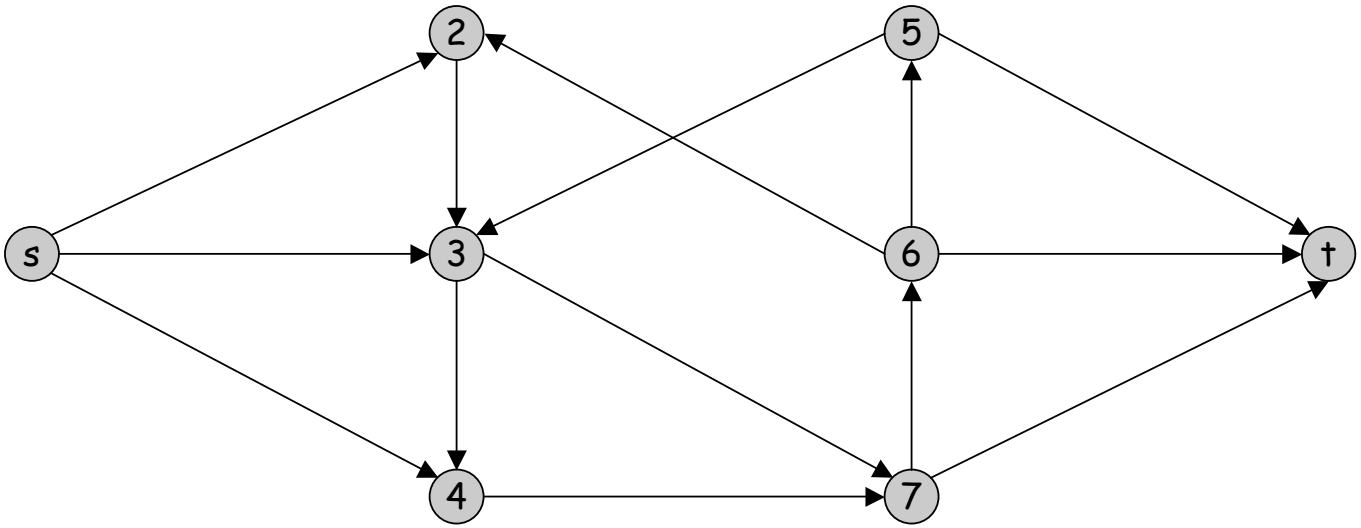
Disjoint Path Problem:

Given a digraph $G = (V, E)$ and two nodes s and t, find the maximum number of edge-disjoint $s - t$ paths.

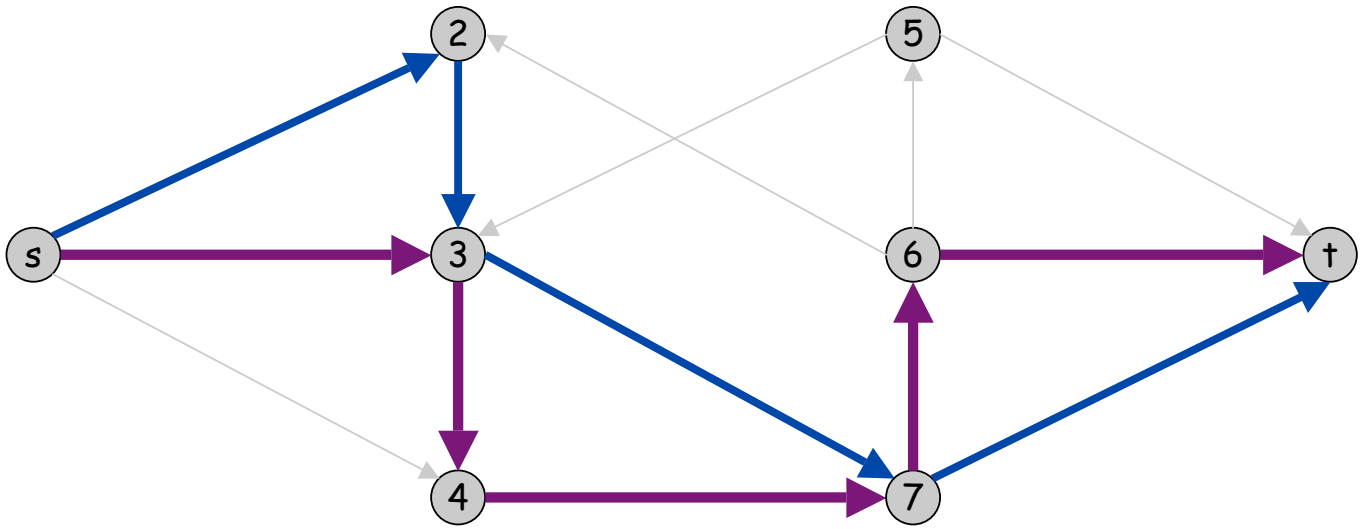
Definition:

Two paths are **edge-disjoint** if they have no edge in common.

Edge Disjoint Paths: example (1)



Edge Disjoint Paths: example (2)



Edge Disjoint Paths: Max-Flow formulation (1)

Simply assign unit capacity to every edge!

Theorem:

The maximum number of edge-disjoint $s - t$ paths equals the max flow value.

Proof.

We prove here $\leq$

- Suppose there are k edge-disjoint paths $P_1, ..., P_k$
- Let $f(e) = 1$ if e participates in some path P_i: else set $f(e) = 0$
- Since path are edge-disjoint, f is a flow of value k

Edge Disjoint Paths: Max-Flow formulation (2)

Theorem:

The maximum number of edge-disjoint $s - t$ paths equals the max flow value.

Proof.

We prove here $\geq$

- Suppose max flow value is k
- Integrality theorem $\Rightarrow$ there exists $0 - 1$ flow of value k
- Consider edge (s, u) with $f(s, u) = 1$
  - by conservation, there exists an edge (u, v) with $f(u, v) = 1$
  - continue until you reach t, always choosing a new edge
- Produces k (not necessarily simple[a]) edge-disjoint paths

□

[a] can eliminate cycles to get simple paths if desired

Network connectivity

The problem:

Given a digraph $G = (V, E)$ and two nodes s and t, find minimum number of edges whose removal disconnects t from s.

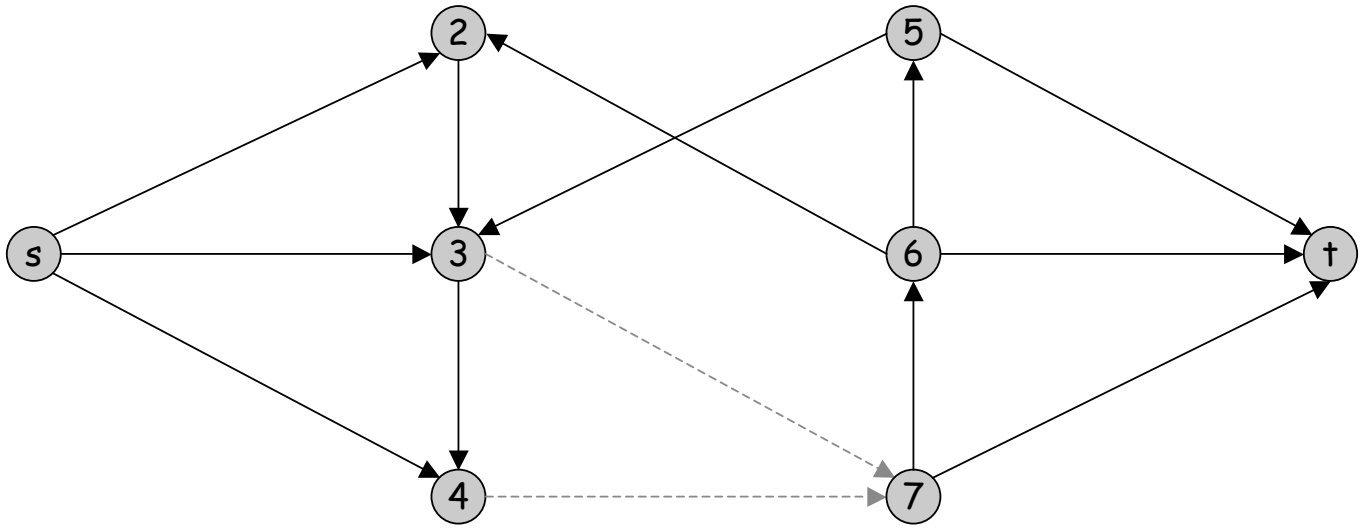
Definition:

A set of edges $F \subseteq E$ disconnects t from s if all $s - t$ paths use at least one edge in F

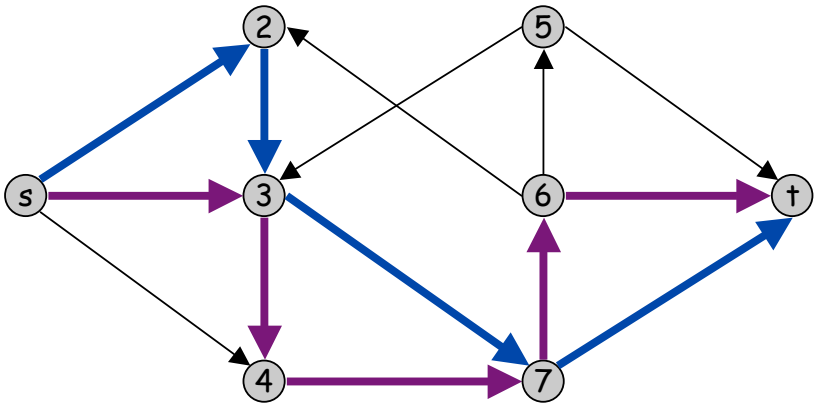
Question[1]: why is this interesting? What are the applications of this problem?

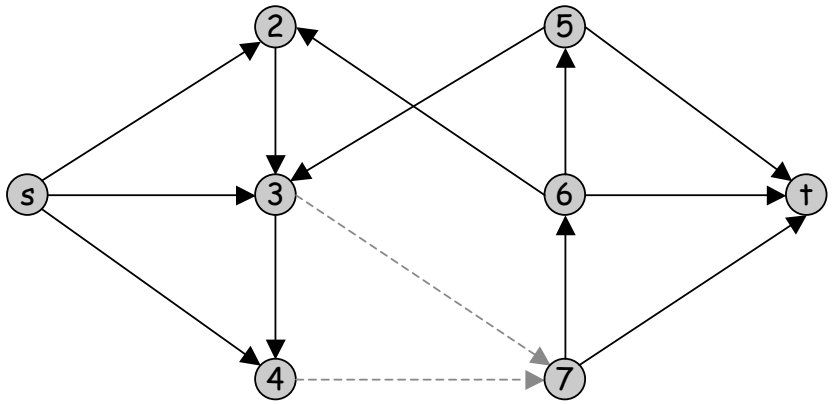
[1] This is a typical exam question!!

Network connectivity: example (1)



Network connectivity: example (2)





Edge-disjoint paths and Network connectivity (1)

Theorem: [Menger 1972]

The max number of edge-disjoint $s - t$ paths is equal to the minimum number of edges whose removal disconnects t from s

Proof.

We start by proving $\leq$

- Suppose the removal of $F \subseteq E$ disconnects t from s, and $|F| = k$
- All $s - t$ paths use at least one edge of F. Hence, the number of edge-disjoint paths is at most k

Edge-disjoint paths and Network connectivity (2)

Theorem: [Menger 1972]

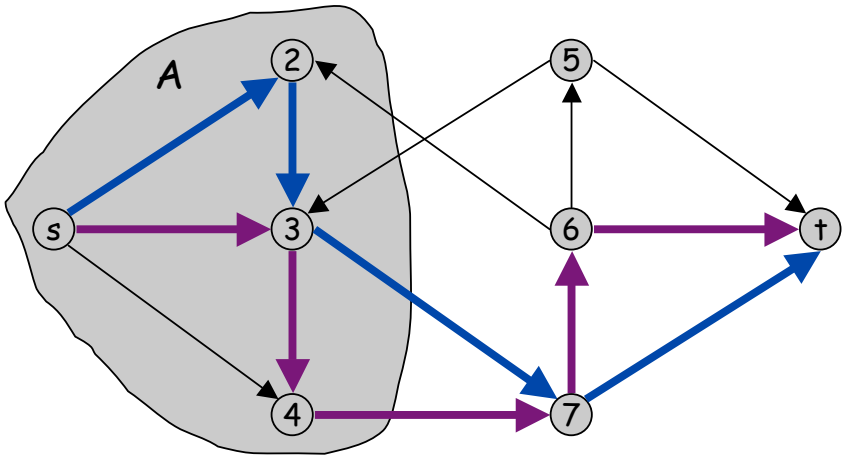
The max number of edge-disjoint $s - t$ paths is equal to the minimum number of edges whose removal disconnects t from s

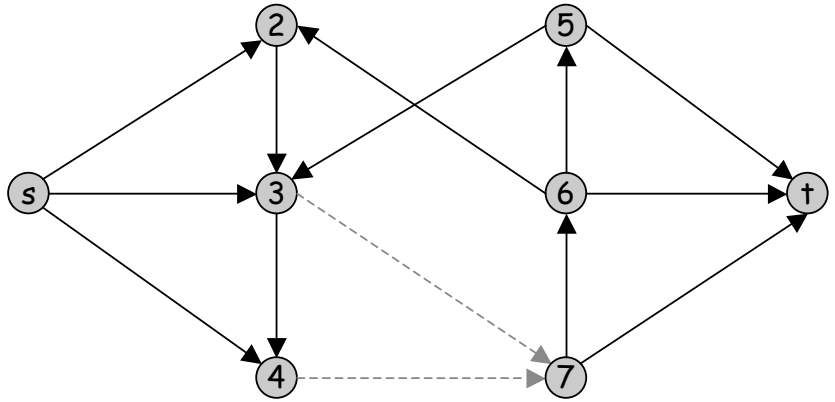
Proof.

We start by proving $\geq$

- Suppose max number of edge-disjoint paths is k
- Then max flow value is k
- Max-flow min-cut $\Rightarrow$ cut (A, B) has capacity k
- Let F be the set of edges going from A to B
- $|F| = k$ and disconnects t from s

Edge-disjoint paths and Network connectivity (2)





Homework

Flow Networks, Max-Flow / Min-Cut problems are fundamental in today's applications!

Your task is simple: read the following research paper:

Research paper:

Randomized decentralized broadcasting algorithms
Laurent Massoulie, Andy Twigg,
Christos Gkantsidis, and Pablo Rodriguez
IEEE INFOCOM 2007

Why? Exam questions could be on this paper; gives a hint to what we'll see in our last lecture.