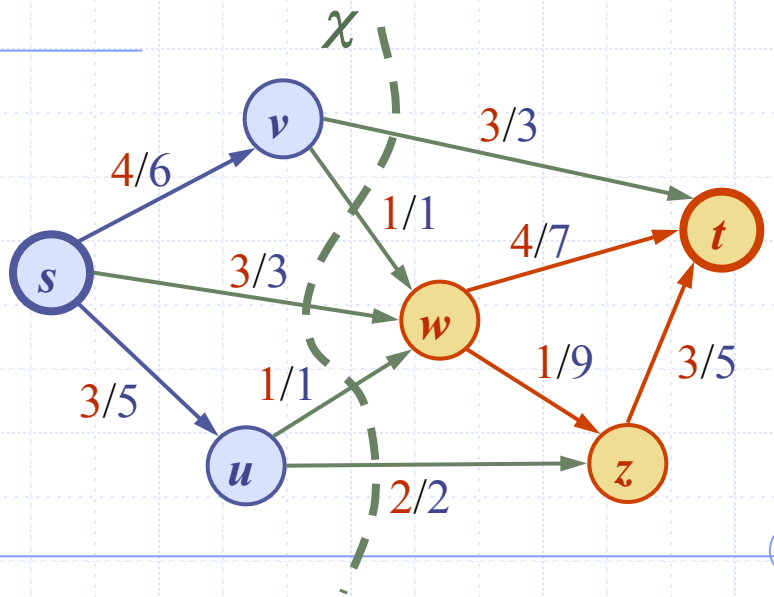
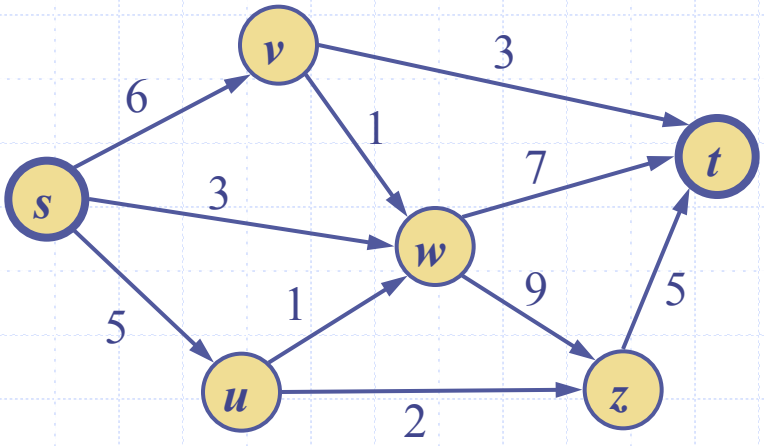
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Maximum Flow



Flow Network

- A flow network (or just network) N consists of
  - A weighted digraph G with nonnegative integer edge weights, where the weight of an edge e is called the capacity $c(e)$ of e
  - Two distinguished vertices, s and t of G, called the source and sink, respectively, such that s has no incoming edges and t has no outgoing edges.
- Example:



Flow

- A flow f for a network N is is an assignment of an integer value $f(e)$ to each edge e that satisfies the following properties:

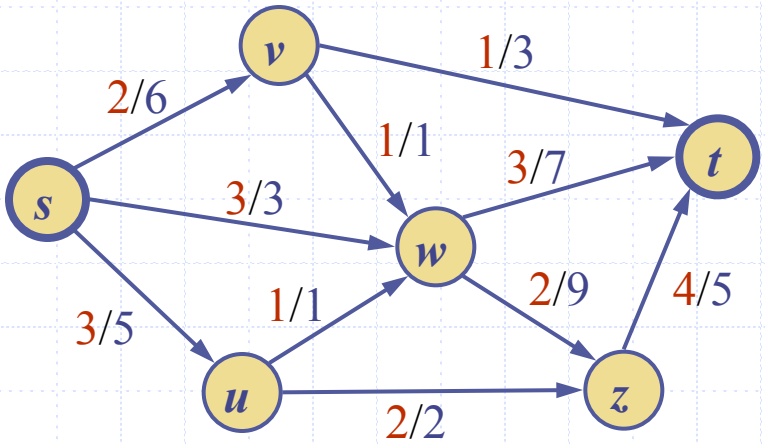
  Capacity Rule: For each edge e, $0 \le f(e) \le c(e)$

  Conservation Rule: For each vertex $v \neq s,t$ $\sum_{e \in E^-(v)} f(e) = \sum_{e \in E^+(v)} f(e)$

  where $E^-(v)$ and $E^+(v)$ are the incoming and outgoing edges of v, resp.

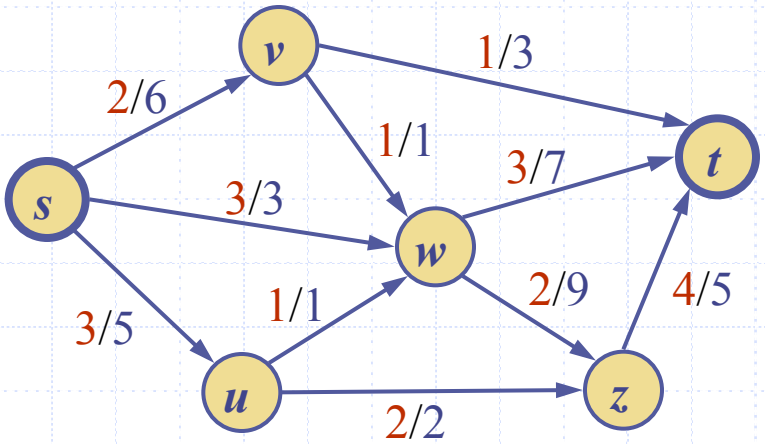
- The value of a flow f, denoted $|f|$, is the total flow from the source, which is the same as the total flow into the sink

- Example:

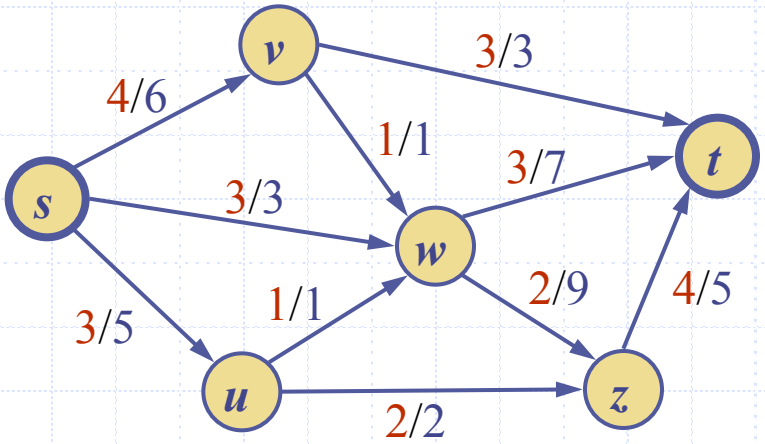


Maximum Flow

- A flow for a network N is said to be maximum if its value is the largest of all flows for N
- The maximum flow problem consists of finding a maximum flow for a given network N
- Applications
  - Hydraulic systems
  - Electrical circuits
  - Traffic movements
  - Freight transportation



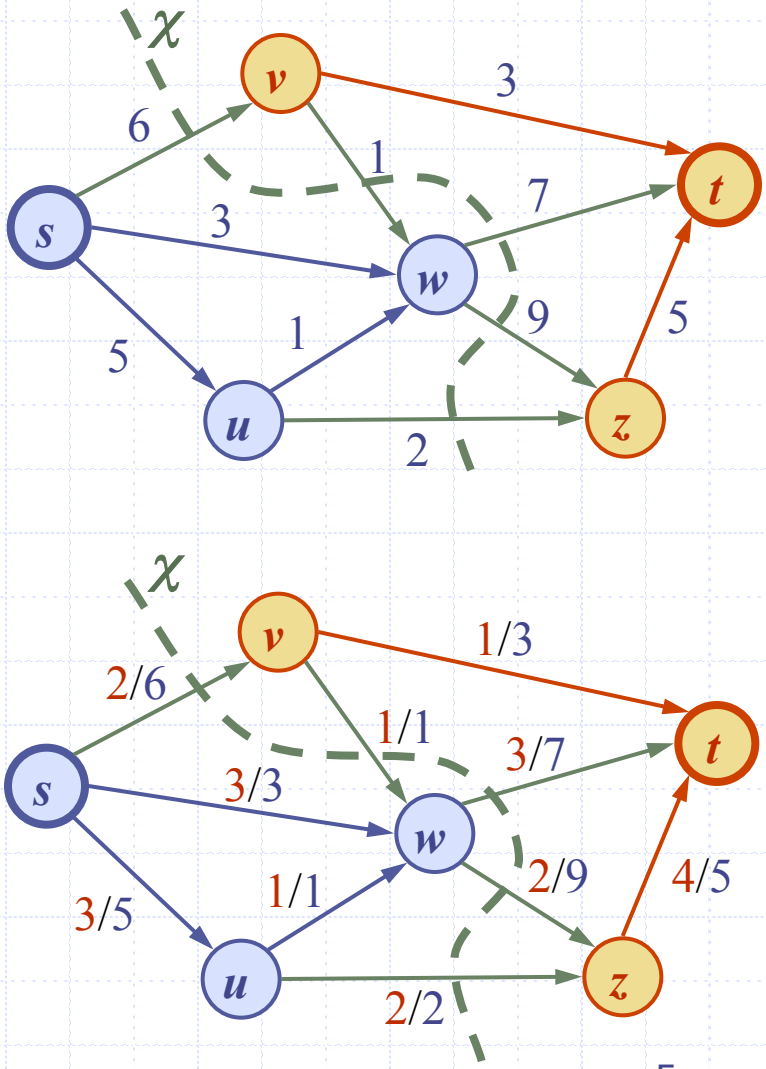
Flow of value 8 = 2 + 3 + 3 = 1 + 3 + 4



Maximum flow of value 10 = 4 + 3 + 3 = 3 + 3 + 4

Cut

- A cut of a network N with source s and sink t is a partition $\chi = (V_s, V_t)$ of the vertices of N such that $s \in V_s$ and $t \in V_t$
  - Forward edge of cut χ: origin in V_s and destination in V_t
  - Backward edge of cut χ: origin in V_t and destination in V_s
- Flow $f(\chi)$ across a cut χ: total flow of forward edges minus total flow of backward edges
- Capacity $c(\chi)$ of a cut χ: total capacity of forward edges
- Example:
  - $c(\chi) = 24$
  - $f(\chi) = 8$



Flows and Cuts

Lemma:

The flow $f(\chi)$ across any cut χ is equal to the flow value $|f|$

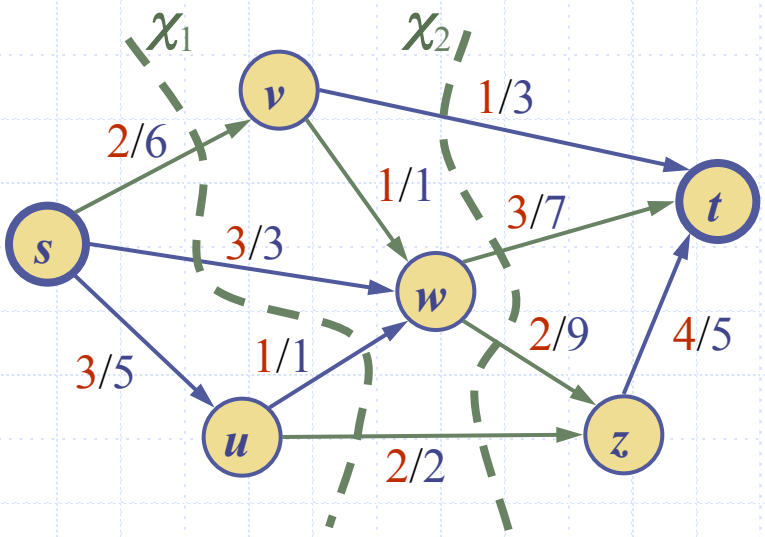
Lemma:

The flow $f(\chi)$ across a cut χ is less than or equal to the capacity $c(\chi)$ of the cut

Theorem:

The value of any flow is less than or equal to the capacity of any cut, i.e., for any flow f and any cut χ, we have

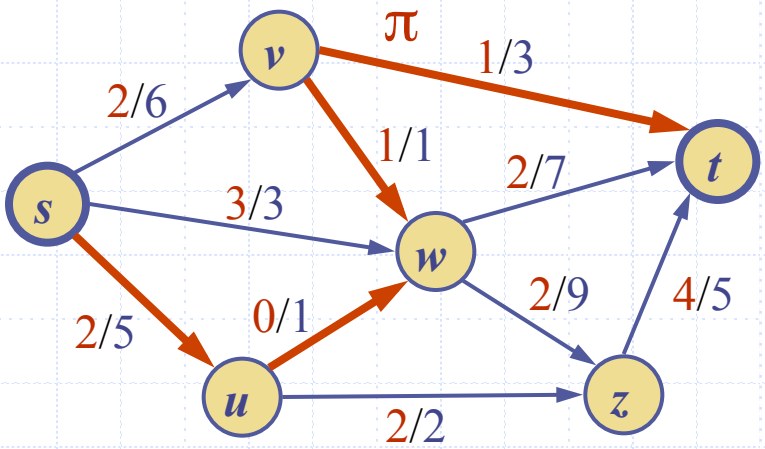
$$|f| \leq c(\chi)$$



$$c(\chi_1) = 12 = 6 + 3 + 1 + 2$$
$$c(\chi_2) = 21 = 3 + 7 + 9 + 2$$
$$|f| = 8$$

Augmenting Path

- Consider a flow f for a network N
- Let e be an edge from u to v:
  - Residual capacity of e from u to v: $\Delta_f(u, v) = c(e) - f(e)$
  - Residual capacity of e from v to u: $\Delta_f(v, u) = f(e)$
- Let π be a path from s to t
  - The residual capacity $\Delta_f(\pi)$ of π is the smallest of the residual capacities of the edges of π in the direction from s to t
- A path π from s to t is an augmenting path if $\Delta_f(\pi) > 0$



$$\Delta_f(s,u) = 3$$
$$\Delta_f(u,w) = 1$$
$$\Delta_f(w,v) = 1$$
$$\Delta_f(v,t) = 2$$
$$\Delta_f(\pi) = 1$$
$$|f| = 7$$

Flow Augmentation

Lemma:

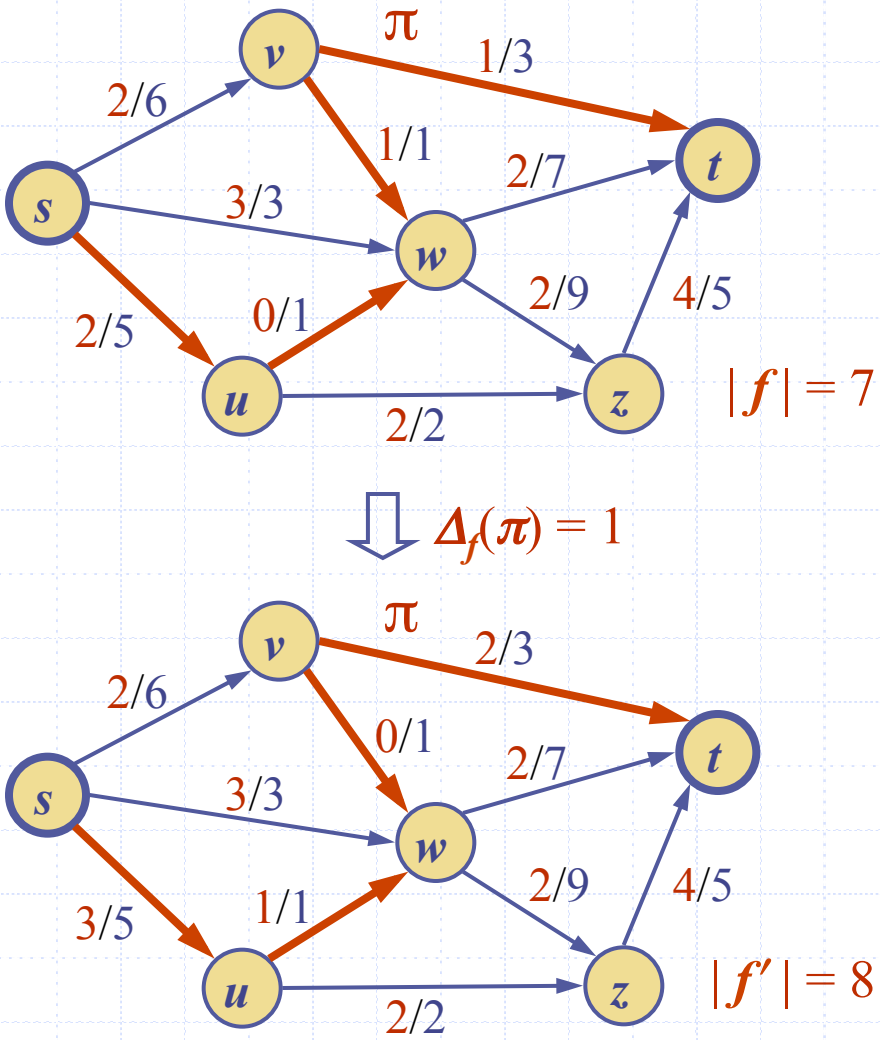
Let π be an augmenting path for flow f in network N. There exists a flow f' for N of value

$$|f'| = |f| + \Delta_f(\pi)$$

Proof:

We compute flow f' by modifying the flow on the edges of π

- Forward edge:
$$f'(e) = f(e) + \Delta_f(\pi)$$
- Backward edge:
$$f'(e) = f(e) - \Delta_f(\pi)$$



The Ford-Fulkerson Algorithm

- Initially, $f(e) = 0$ for each edge e
- Repeatedly
  - Search for an augmenting path π
  - Augment by $\Delta_f(\pi)$ the flow along the edges of π
- A specialization of DFS (or BFS) searches for an augmenting path
  - An edge e is traversed from u to v provided $\Delta_f(u, v) > 0$

Algorithm MaxFlowFordFulkerson(N):
Input: Flow network $N = (G, c, s, t)$
Output: A maximum flow f for N

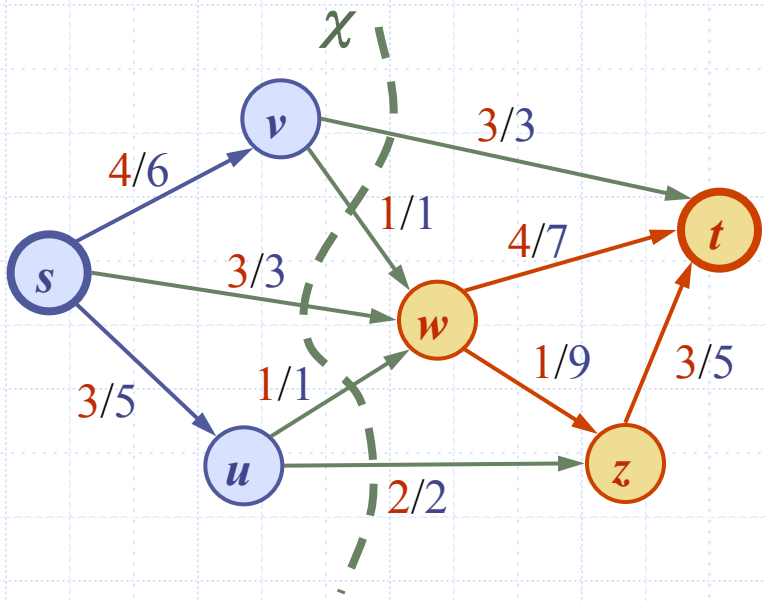
for each edge $e \in N$ **do**
  $f(e) \leftarrow 0$
stop $\leftarrow$ **false**
repeat
  traverse G starting at s to find an augmenting path for f
  if an augmenting path π exists **then**
    // Compute the residual capacity $\Delta_f(\pi)$ of π
    $\Delta \leftarrow +\infty$
    for each edge $e \in \pi$ **do**
      if $\Delta_f(e) < \Delta$ **then**
        $\Delta \leftarrow \Delta_f(e)$
    for each edge $e \in \pi$ **do** // push $\Delta = \Delta_f(\pi)$ units along π
      if e is a forward edge **then**
        $f(e) \leftarrow f(e) + \Delta$
      else
        $f(e) \leftarrow f(e) - \Delta$ // e is a backward edge
  else
    stop $\leftarrow$ **true** // f is a maximum flow
until *stop*

Max-Flow and Min-Cut

- Termination of Ford-Fulkerson's algorithm
  - There is no augmenting path from s to t with respect to the current flow f
- Define
  - V_s set of vertices reachable from s by augmenting paths
  - V_t set of remaining vertices
- Cut $\chi = (V_s, V_t)$ has capacity $c(\chi) = |f|$
  - Forward edge: $f(e) = c(e)$
  - Backward edge: $f(e) = 0$
- Thus, flow f has maximum value and cut χ has minimum capacity

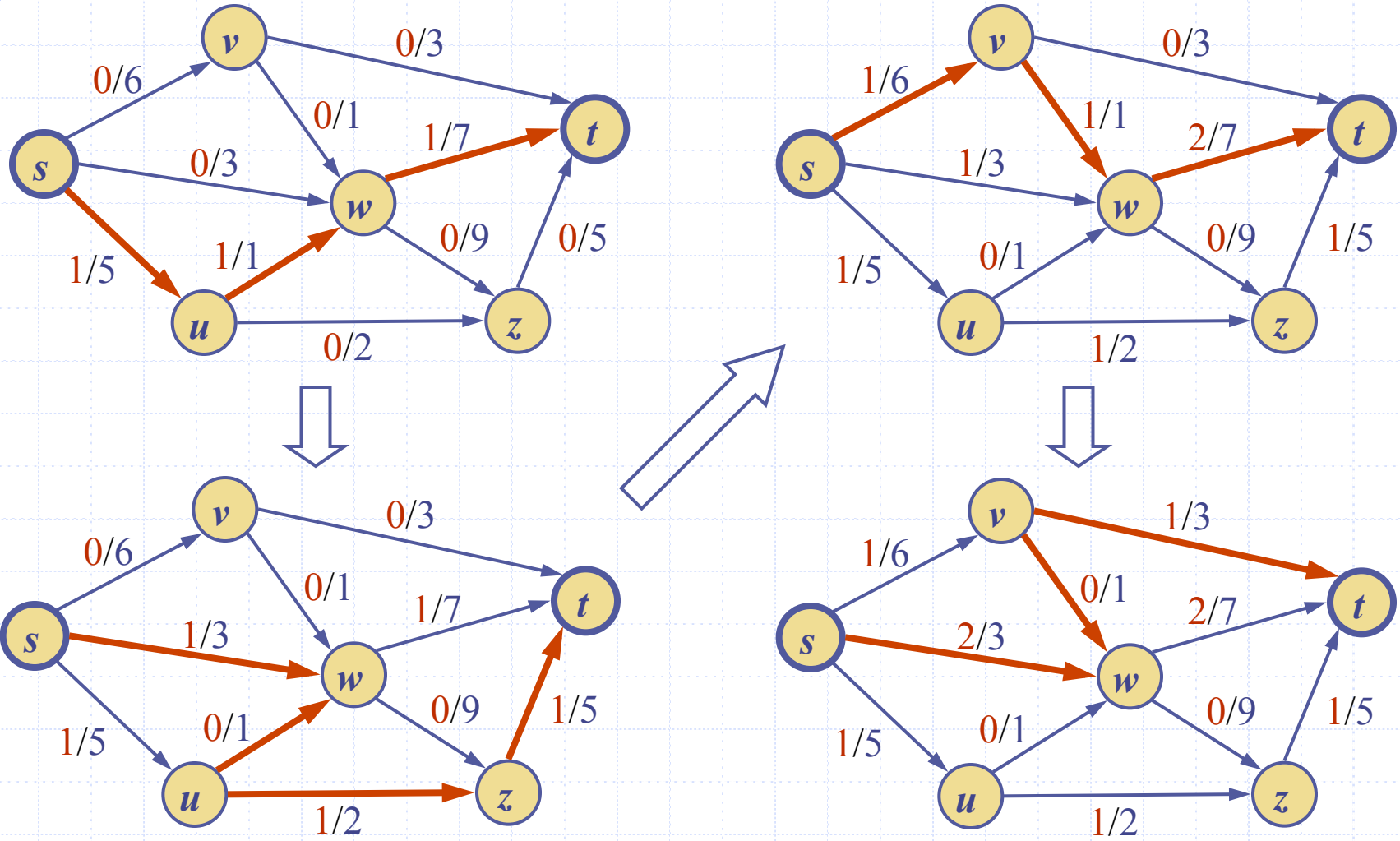
Theorem:

The value of a maximum flow is equal to the capacity of a minimum cut

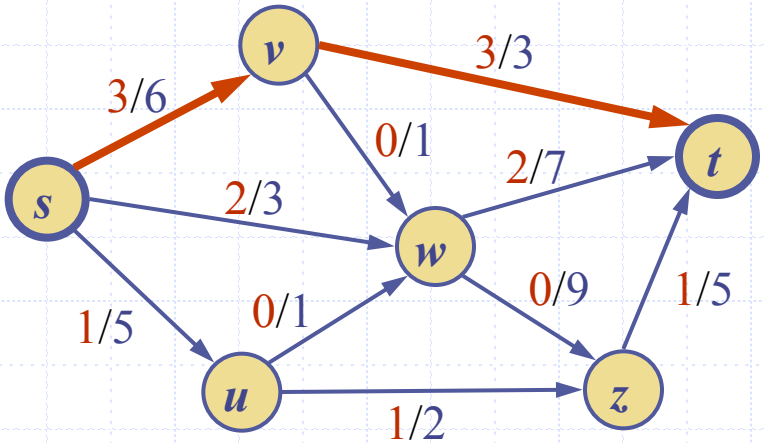


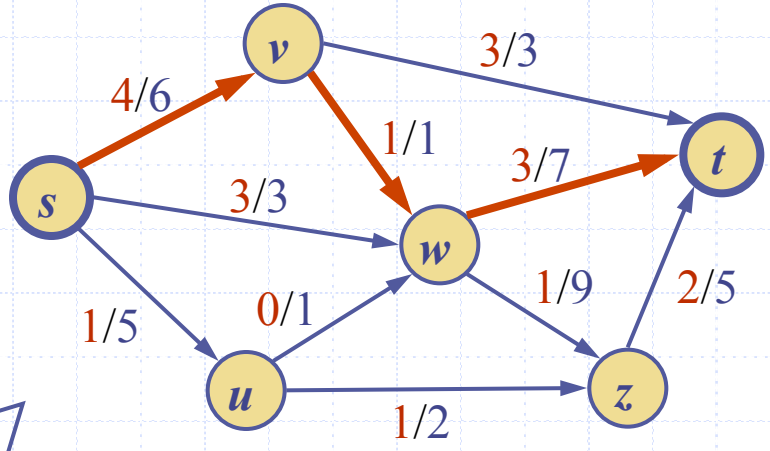
$c(\chi) = |f| = 10$

Example (1)

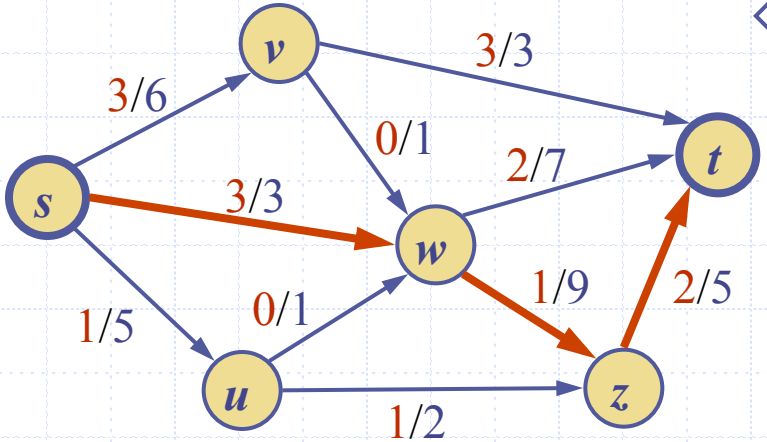

v
0/3
0/6
0/1
t
1/7
s
0/3
w
1/5
1/1
0/9
0/5
u
z
0/2
v
0/3
1/6
1/1
t
2/7
s
1/3
w
1/5
0/1
0/9
1/5
u
z
1/2
v
0/3
0/6
0/1
t
1/7
s
1/3
w
1/5
0/1
0/9
1/5
u
z
1/2
v
1/3
1/6
0/1
t
2/7
s
2/3
w
1/5
0/1
0/9
1/5
u
z
1/2

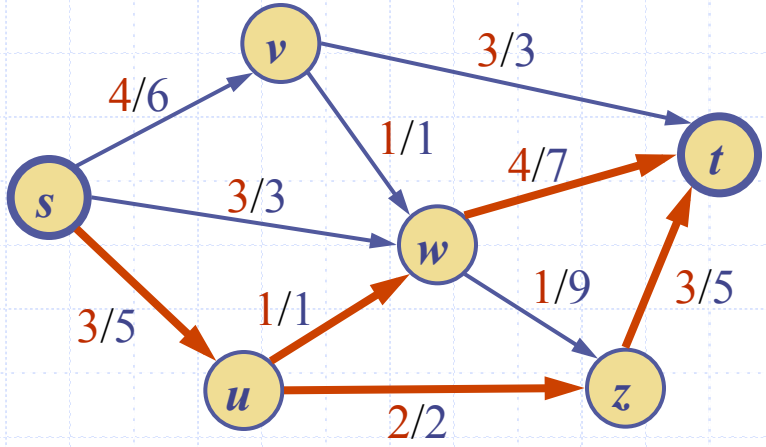
Example (2)





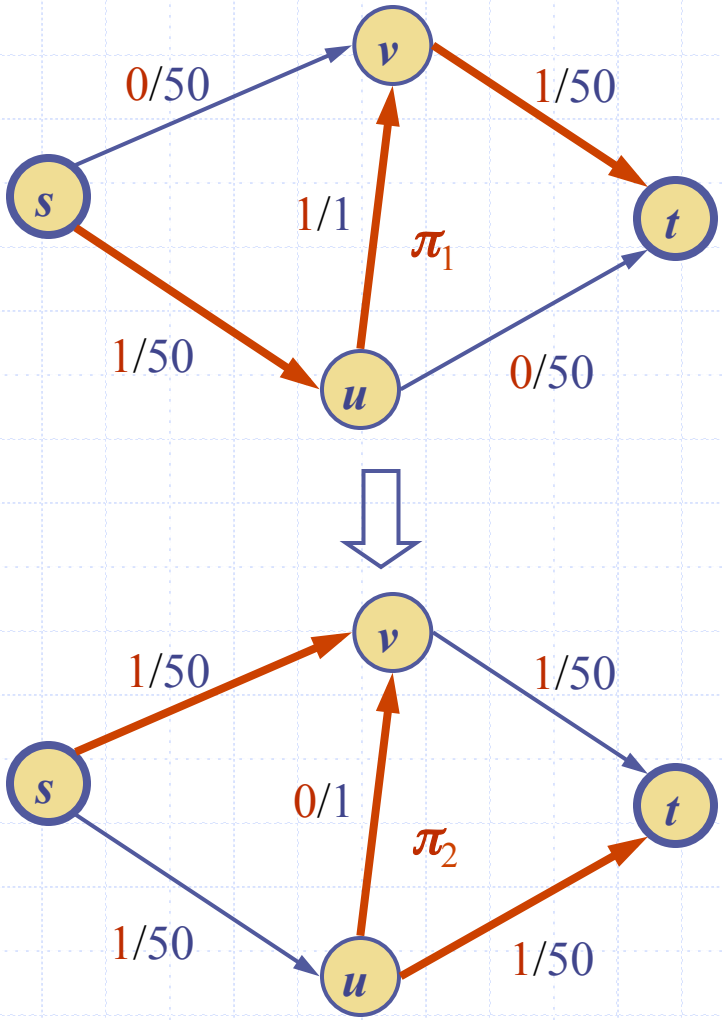
two steps





Analysis

- In the worst case, Ford-Fulkerson's algorithm performs $|f^*|$ flow augmentations, where f^* is a maximum flow
- Example
  - The augmenting paths found alternate between π_1 and π_2
  - The algorithm performs 100 augmentations
- Finding an augmenting path and augmenting the flow takes $O(n + m)$ time
- The running time of Ford-Fulkerson's algorithm is $O(|f^*|(n + m))$



Maximum Bipartite Matching

- In the maximum bipartite matching problem, we are given a connected undirected graph with the following properties:
  - The vertices of G are partitioned into two sets, X and Y.
  - Every edge of G has one endpoint in X and the other endpoint in Y.
- Such a graph is called a **bipartite graph**.
- A **matching** in G is a set of edges that have no endpoints in common—such a set "pairs" up vertices in X with vertices in Y so that each vertex has at most one "partner" in the other set.
- The maximum bipartite matching problem is to find a matching with the greatest number of edges.

Reduction to Max Flow

Let G be a bipartite graph whose vertices are partitioned into sets X and Y. We create a flow network H such that a maximum flow in H can be immediately converted into a maximum matching in G:

- We begin by including all the vertices of G in H, plus a new source vertex s and a new sink vertex t.
- Next, we add every edge of G to H, but direct each such edge so that it is oriented from the endpoint in X to the endpoint in Y. In addition, we insert a directed edge from s to each vertex in X, and a directed edge from each vertex in Y to t. Finally, we assign to each edge of H a capacity of 1.

- Given a flow f for H, we use f to define a set M of edges of G using the rule that an edge e is in M whenever f(e) = 1.

Example and Analysis

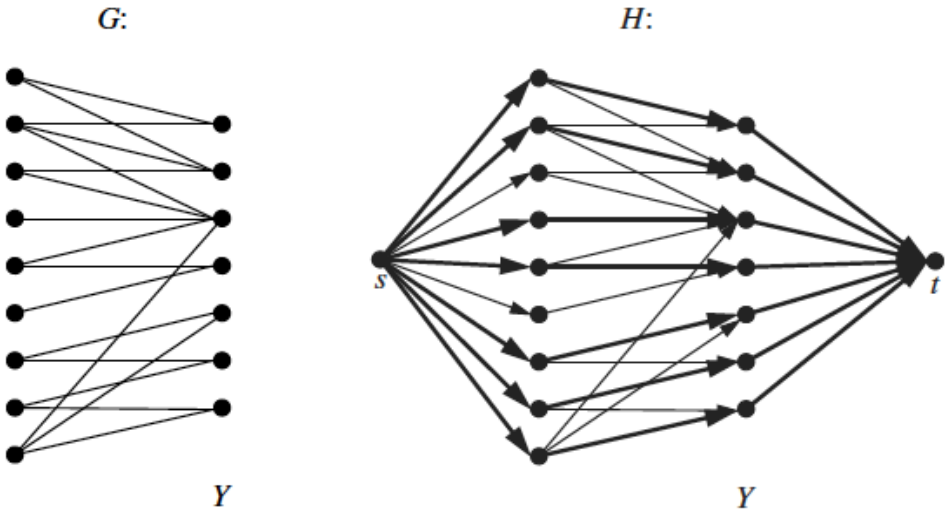


Figure 16.11: (a) A bipartite graph G. (b) Flow network H derived from G and a maximum flow in H; thick edges have unit flow and other edges have zero flow.

- Running time is O(nm), because G is connected.

Baseball Elimination

- Let T be a set of teams in a sports league, which, for historical reasons, let us assume is baseball.
- At any point during the season, each team, i, in T, will have some number, w_i, of wins, and will have some number, g_i, of games left to play.
- The **baseball elimination problem** is to determine whether it is possible for team i to finish the season in first place, given the games it has already won and the games it has left to play.
- Note that this depends on more than just the number of games left for team i, however; it also depends on the respective schedules of team i and the other teams.

Baseball Elimination Example

- Let $g_{i,j}$ denote the number of games remaining between team i and team j, so that g_i is the sum, over all j, of the $g_{i,j}$'s.

Team i	Wins w_i	Games Left g_i	Schedule $(g_{i,j})$ LA	Oak	Sea	Tex
Los Angeles	81	8	-	1	6	1
Oakland	77	4	1	-	0	3
Seattle	76	7	6	0	-	1
Texas	74	5	1	3	1	-

Table 16.12: A set of teams, their standings, and their remaining schedule. Clearly, Texas is eliminated from finishing in first place, since it can win at most 79 games. In addition, even though it is currently in second place, Oakland is also eliminated, because it can win at most 81 games, but in the remaining games between LA and Seattle, either LA wins at least 1 game and finishes with at least 82 wins or Seattle wins 6 games and finishes with at least 82 wins.

Reduction to Max Flow

With all the different ways for a team, k, to be eliminated, it might at first seem like it is computationally infeasible to determine whether team k is eliminated. Still, we can solve this problem by a reduction to a network flow problem. Let T' denote the set of teams other than k, that is, $T' = T - \{k\}$. Also, let L denote the set of games that are left to play among teams in T', that is,

$$L = \{\{i, j\}: \ i, j \in T' \text{ and } g_{i,j} > 0\}.$$

Finally, let W denote the largest number of wins that are possible for team k given the current standings, that is, $W = w_k + g_k$.

- Let us assume no single team eliminates team k (since this is easy to check).

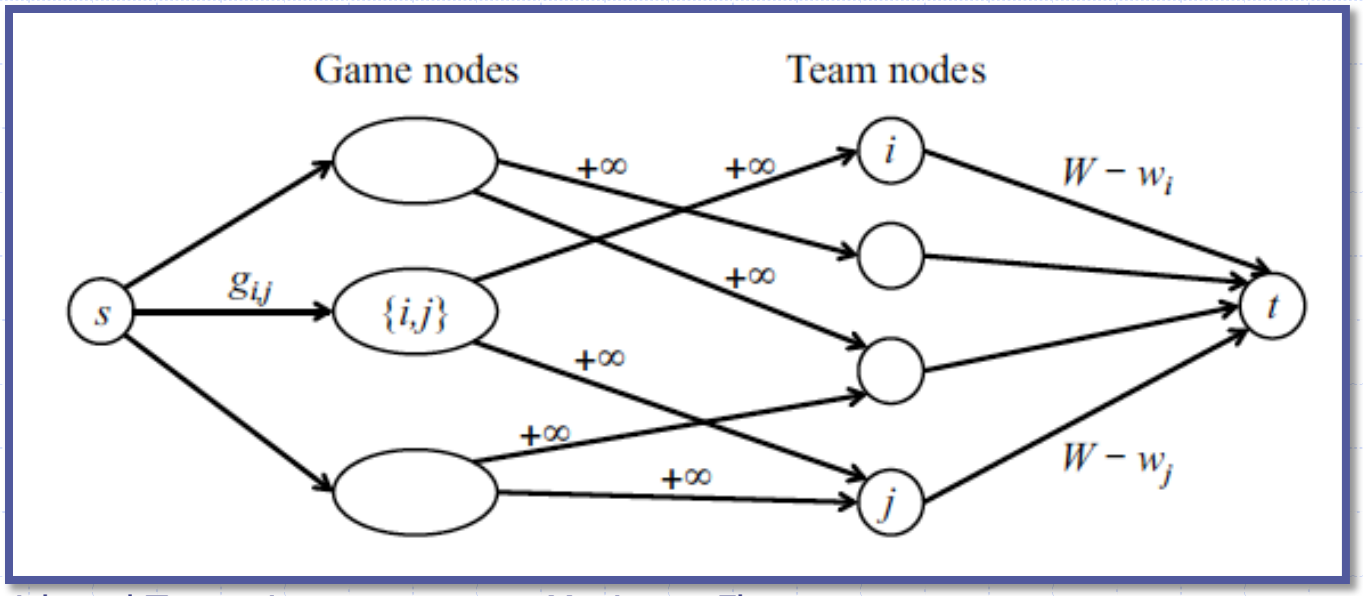
Creating the Graph

- To consider how a combination of teams and game outcomes might eliminate team k, we create a graph, G, that has as its vertices a source, s, a sink, t, and the sets T′ and L. Then, let us include the following edges in G:

- For each game pair, $\{i, j\}$, in L, add an edge $(s, \{i, j\})$, and give it capacity $g_{i,j}$.
- For each game pair, $\{i, j\}$, in L, add edges $(\{i, j\}, i)$ and $(\{i, j\}, j)$, and give these edges capacity $+\infty$.
- For each team, i, add an edge (i, t) and give it capacity $W - w_i$, which cannot be negative in this case, since we ruled out the case when $W < w_i$.

Creating the Graph, Example

- For each game pair, $\{i, j\}$, in L, add an edge $(s, \{i, j\})$, and give it capacity $g_{i,j}$.
- For each game pair, $\{i, j\}$, in L, add edges $(\{i, j\}, i)$ and $(\{i, j\}, j)$, and give these edges capacity $+\infty$.
- For each team, i, add an edge (i, t) and give it capacity $W - w_i$, which cannot be negative in this case, since we ruled out the case when $W < w_i$.



Intuition and Analysis

The intuition behind the construction for G is that wins flow out from the source, s, are split at each game node, $\{i, j\}$, to allocate wins between each pair of teams, i and j, and then are absorbed by the sink, t. The flow on each edge, $(\{i, j\}, i)$, represents the number of games in which team i beats j, and the flow on each edge, (i, t), represents the number of remaining games that could be won by team i. Thus, maximizing the flow in G is equivalent to testing if it is possible to allocate wins among all the remaining games not involving team k so that no team goes above W wins. So we compute a maximum flow for G.

- We can solve baseball elimination for any team in a set of n teams by solving a single maximum flow problem on a network with at most $O(n^2)$ vertices and edges.