CSE 421
Introduction to Algorithms

Lecture 18: Applications/Extensions of Network Flow

Announcements

This week:

Tomorrow 4:30 pm: Zoom review session for Q&A. Bring your questions.

- Zoom link TBA.

Wednesday: No lecture. **Midterm 6:00 – 7:30 pm**

- HW6 out

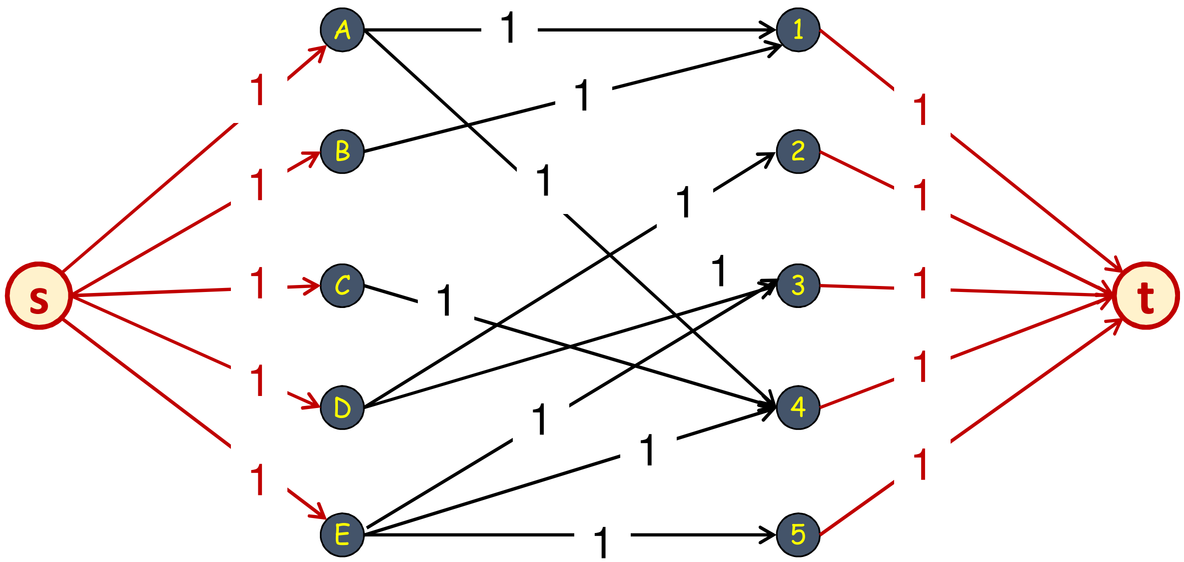
Thursday: Section Network Flow

Friday: Holiday

Recall: Bipartite Matching using Network Flow

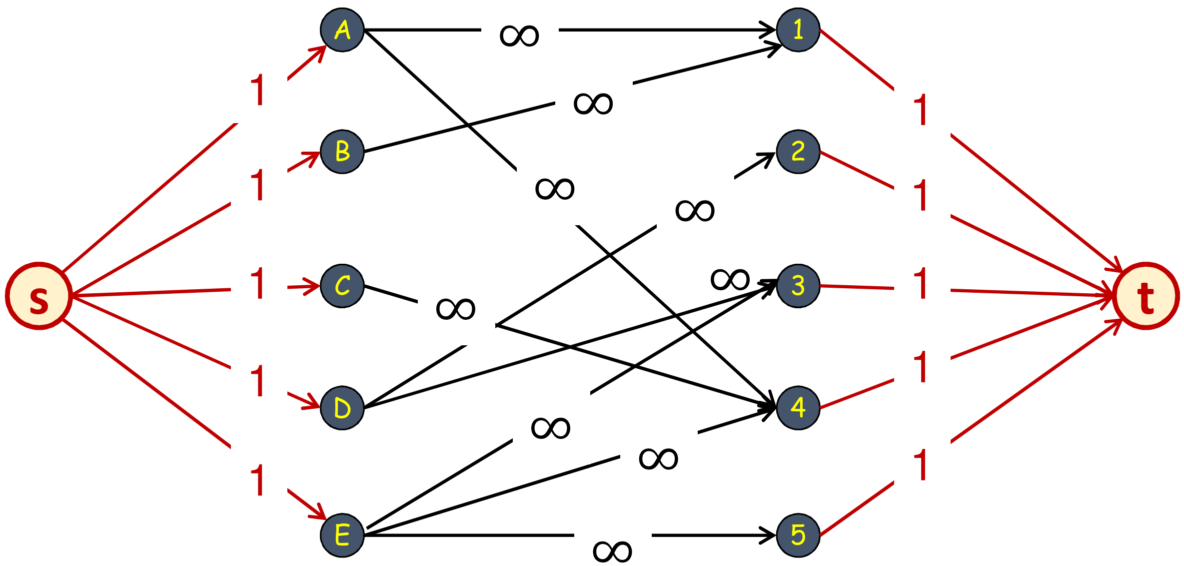
Add new source **s** pointing to left set, new sink **t** pointed to by right set.

Direct all edges from left to right with capacity 1. Compute MaxFlow.



More Bipartite Matching using Network Flow

It also works if we have no capacity limit on the edges of the input graph G since we can never get more than 1 unit of flow to these edges and flows are integral w.l.o.g.



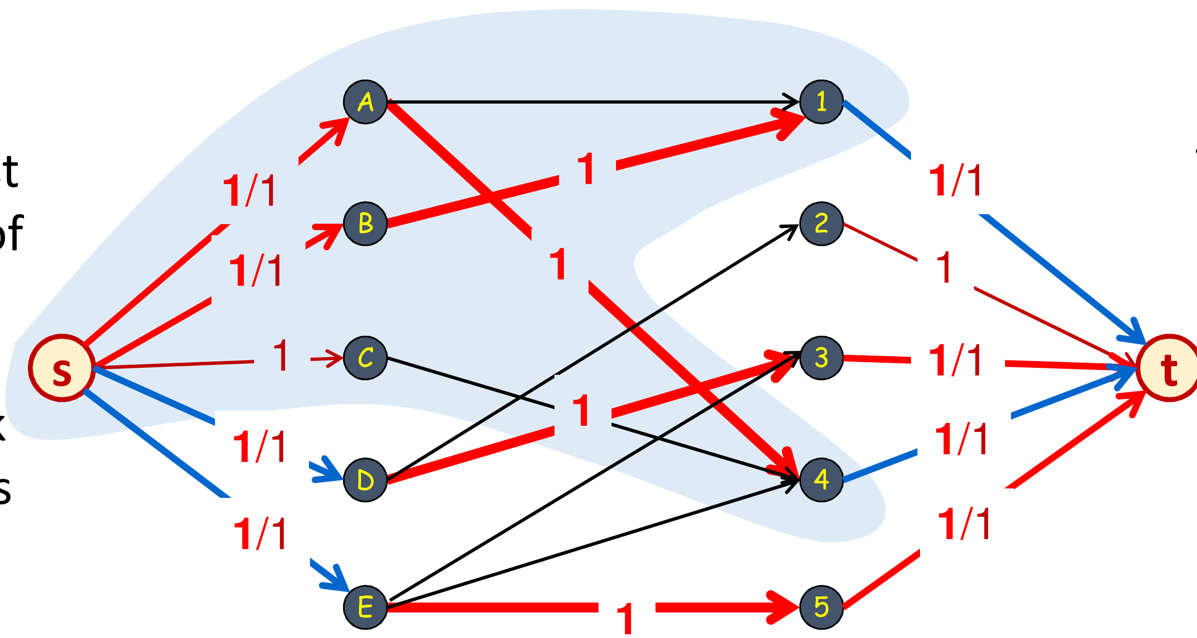
Bipartite Matching using Network Flow

Add new source **s** pointing to left set, new sink **t** pointed to by right set.

Direct edges left to right; new edges have capacity 1. Compute MaxFlow.

Correctness:
Integer flow just gives a subset of edges.

Source and sink edges imply it is a matching



Time $O(mn)$

Optimality

Perfect Matching

Defn: A matching $M \subseteq E$ is perfect iff every vertex is in some edge.

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

- Clearly we must have $|L| = |R|$.
- What other conditions are necessary?
- What conditions are sufficient?

Perfect Matching

Notation: For S be a set of vertices let $N(S)$ be the set of vertices adjacent to nodes in S (the "neighborhood of 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 a different node in $N(S)$. ■

Hall's Theorem say this is the only condition we need: If there is no perfect matching then there is some subset $S \subseteq L$ with $|N(S)| < |S|$.

Hall's Theorem Proof

No perfect matching

$\Rightarrow$ MaxFlow value $< |L|$

$\Rightarrow$ MinCut value $< \ |L|$.

Let (A, B) be cut with $c(A, B) < L$

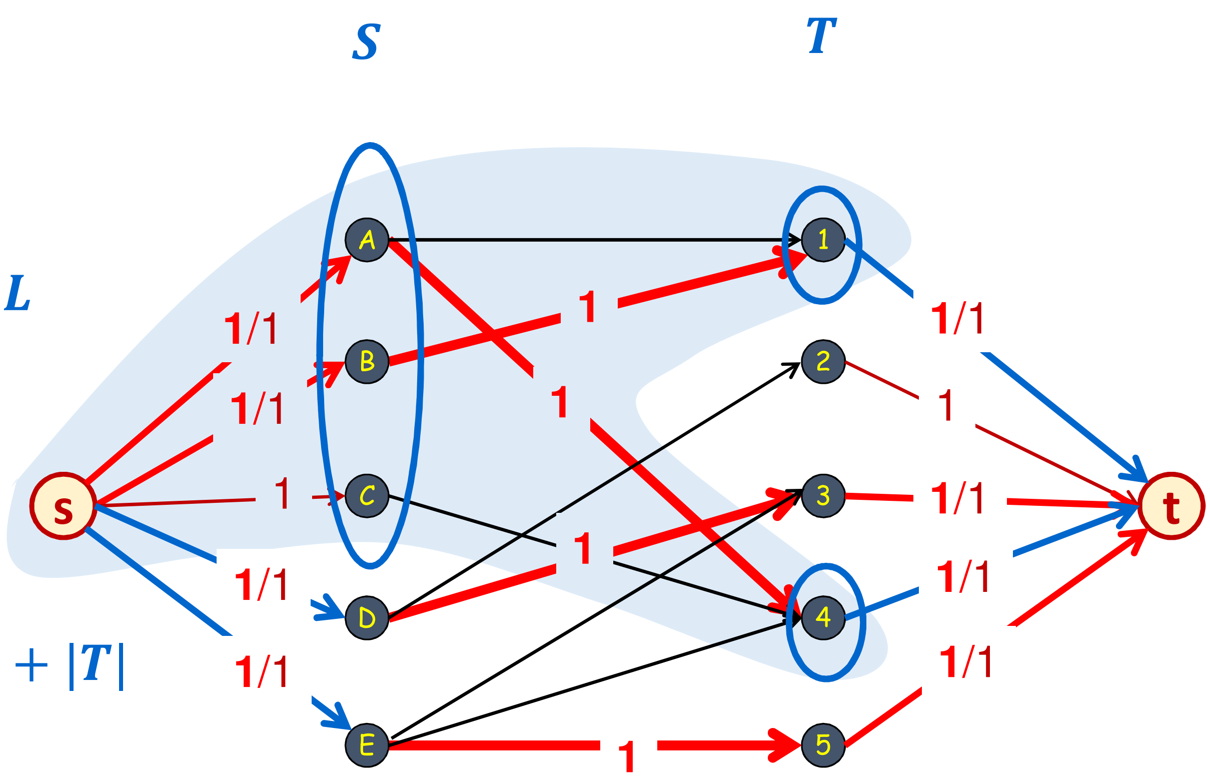
Let $S = A \cap L$ and $T = A \cap R$.

Must have $N(S) \subseteq T$

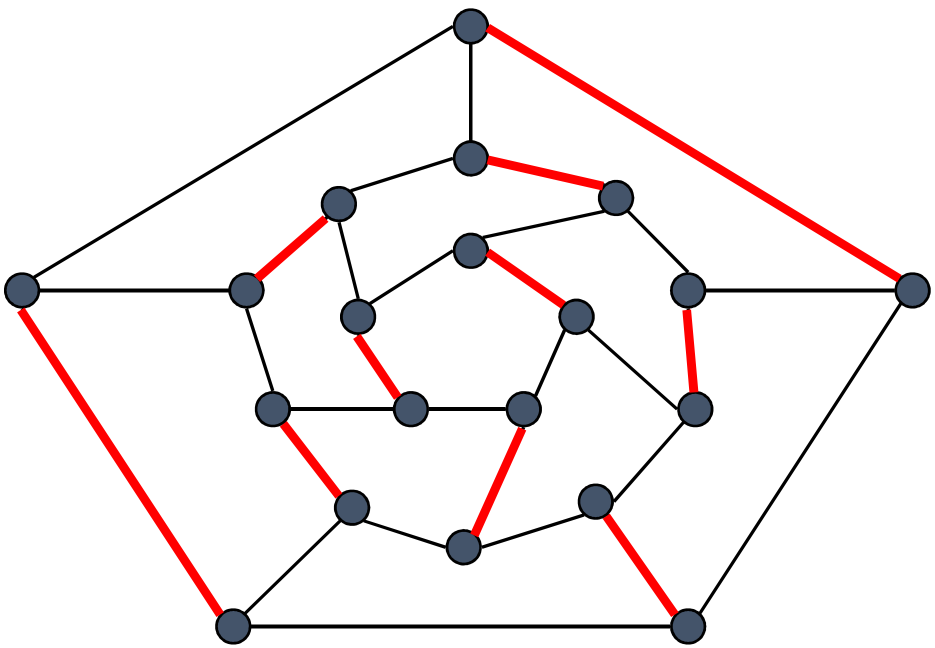
since $c(A, B)$ is finite.

(no edges of G can cross cut)

Then $|L| > \ c(A, B) \ = \ |L| - |S| + |T|$
so $|N(S)| \leq |T| < |S|$. ∎



Matching in General Graphs?



Matching: Best Running Times

Bipartite matching running times?

- Generic augmenting path: $O(mn)$.
- Shortest augmenting path: $O(mn^{1/2})$.
- Until very recently these were the best...
- Recent algorithms for maxflow give $O(m^{1+o(1)})$ time with high probability.

General matching?

- Augmenting paths don't work
- [Edmonds 1965] Added notion of "blossoms" for first polytime algorithm $O(n^4)$
  - One of the most famous/important papers in the field: "Paths, Trees, and Flowers"
- [Micali-Vazirani 1980, 2020] Tricky data structures and analysis. $O(mn^{1/2})$

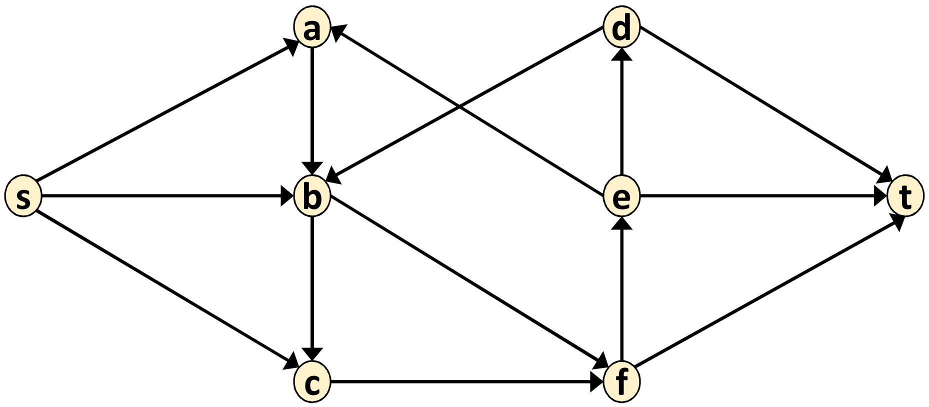
Disjoint Paths

Edge-Disjoint Paths

Defn: Two paths in a graph are edge-disjoint iff they have no edge in common.

Disjoint path problem: **Given:** a directed graph $G = (V, E)$ and two vertices s and t.
Find: the maximum # of edge-disjoint s-t simple paths in G.

Application: Routing in communication networks.

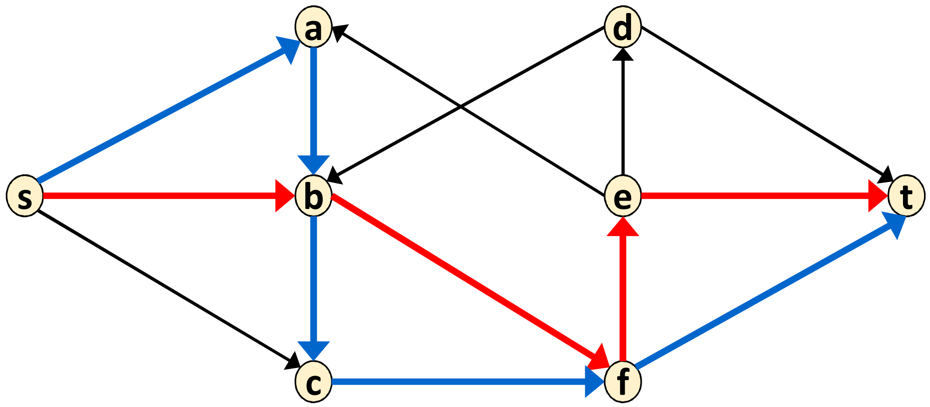


Edge-Disjoint Paths

Defn: Two paths in a graph are edge-disjoint iff they have no edge in common.

Disjoint path problem: **Given:** a directed graph $G = (V, E)$ and two vertices s and t.
Find: the maximum # of edge-disjoint simple s-t paths in G.

Application: Routing in communication networks.

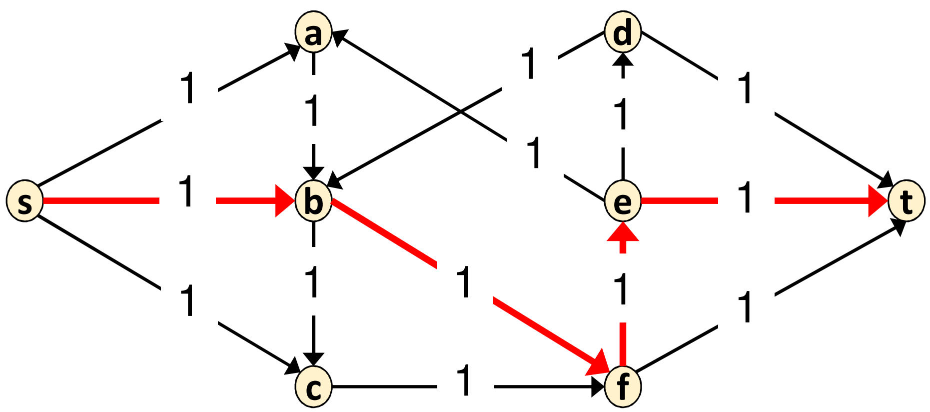


Edge-Disjoint Paths

MaxFlow for edge-disjoint paths

- Delete edges into s or out of t
- Assign capacity 1 to every edge
- Compute MaxFlow

Theorem: MaxFlow = # edge-disjoint paths



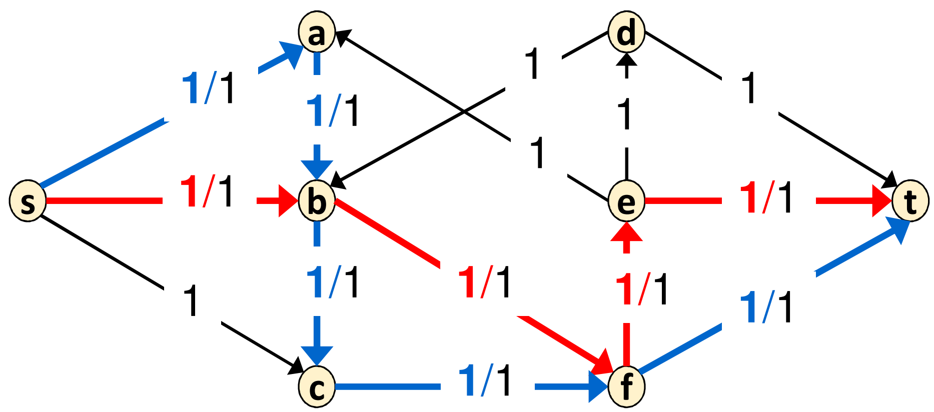
Edge-Disjoint Paths

MaxFlow for edge-disjoint paths

- Delete edges into s or out of t
- Assign capacity **1** to every edge
- Compute MaxFlow

Theorem: MaxFlow = # edge-disjoint paths

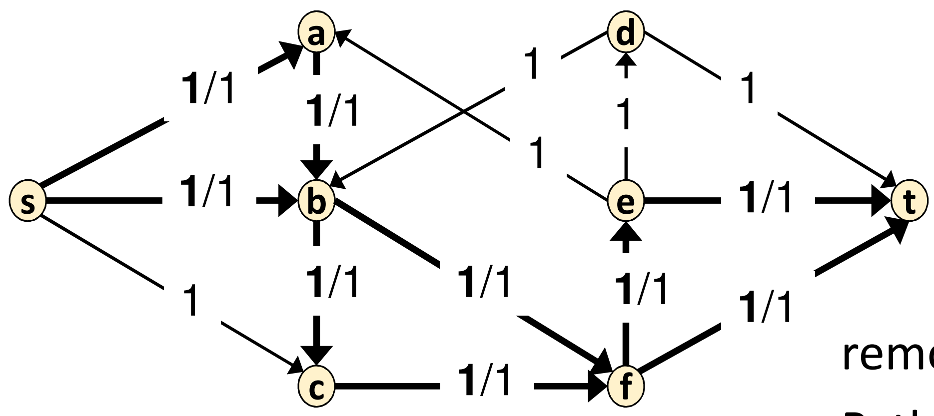
Proof: $\geq$: Assign flow 1 to each edge in the set of paths



Edge-Disjoint Paths

MaxFlow for edge-disjoint paths

- Delete edges into s or out of t
- Assign capacity $\mathbf{1}$ to every edge
- Compute MaxFlow



Theorem: MaxFlow $=$ # edge-disjoint paths

Proof: $\geq$: Assign flow 1 to each edge in the set of paths

$\leq$: Consider any integral maximum flow f on G

By integrality, each edge with flow has flow 1.

Remove any directed cycles in f with flow; still have a maxflow.

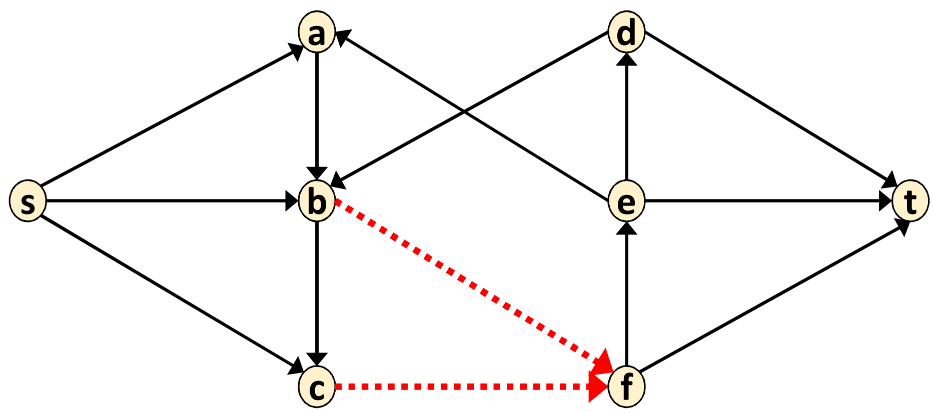
Greedily choose s-t paths, one by one, removing candidate flow edge after using it.

Paths are simple since no directed cycles. ■

Network Connectivity

Defn: A set of edges $F \subseteq E$ in $G = (V, E)$ disconnects t from s iff every s-t path uses at least one edge in F. (Equivalently, removing all edges in F makes t unreachable.)

Network Connectivity: **Given:** a directed graph $G = (V, E)$ and two nodes s and t,
Find: minimum # of edges whose removal disconnects t from s.

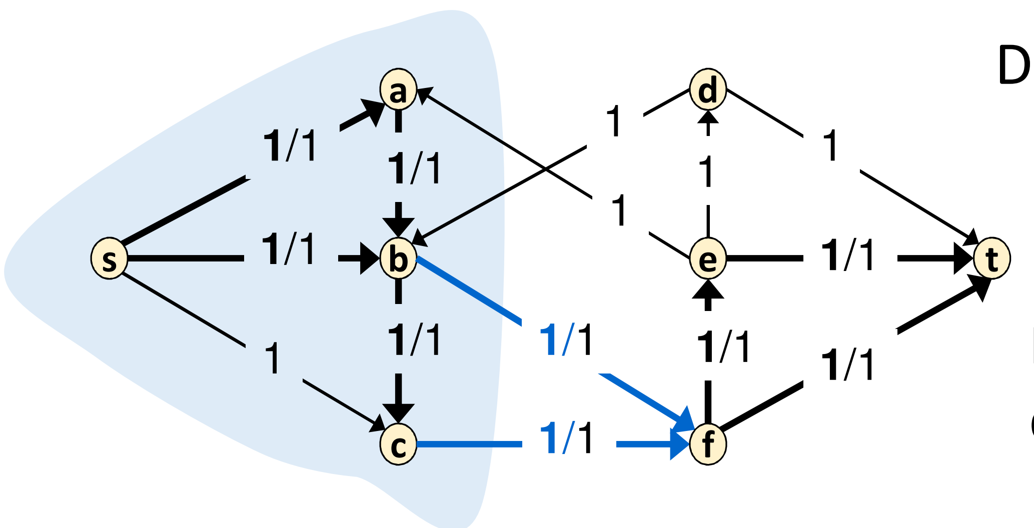


Min # of disconnecting edges: **2**
No s-t path remains.

Edge-Disjoint Paths and Network Connectivity

Menger's Theorem: Maximum # of edge-disjoint s-t paths
$=$ Minimum # of edges whose removal disconnects t from s.

Proof: Choose maximum set of MaxFlow edge-disjoint s-t paths.



Disconnecting set needs
$\geq$ 1 edge from each path
$=$ MaxFlow $=$ MinCut edges.

Edges out of minimum cut is a disconnecting set of size MinCut ∎

Edge-Disjoint Paths in Undirected Graphs

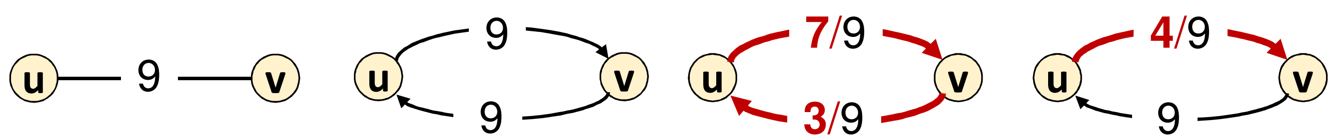
Both # of edge-disjoint paths and disconnecting sets make sense for an undirected graph $G = (V, E)$, too. Same ideas work:

- Replace each undirected edge $\{u, v\}$ with directed edges (u, v) and (v, u) to get directed graph $G' = (V, E')$ and run directed graph algorithm on G'.

- After removing directed cycles, flow can use only one of (u, v) or (v, u).
- Include edge $\{u, v\}$ on a path if either one is used in directed version.

The same idea works in general for Network Flow on undirected graphs:

- Remove flow cycles:



Circulation with Demands

Circulation with Demands

- Single commodity, directed graph $G = (V, E)$
- Each node v has an associated demand $d(v)$
  - Needs to receive an amount of the commodity: demand $d(v) > 0$
  - Supplies some amount of the commodity: "demand" $d(v) < 0$ (amount = $|d(v)|$)
- Each edge e has a capacity $c(e) \geq 0$.
- Nothing lost: $\sum_v d(v) = 0$.

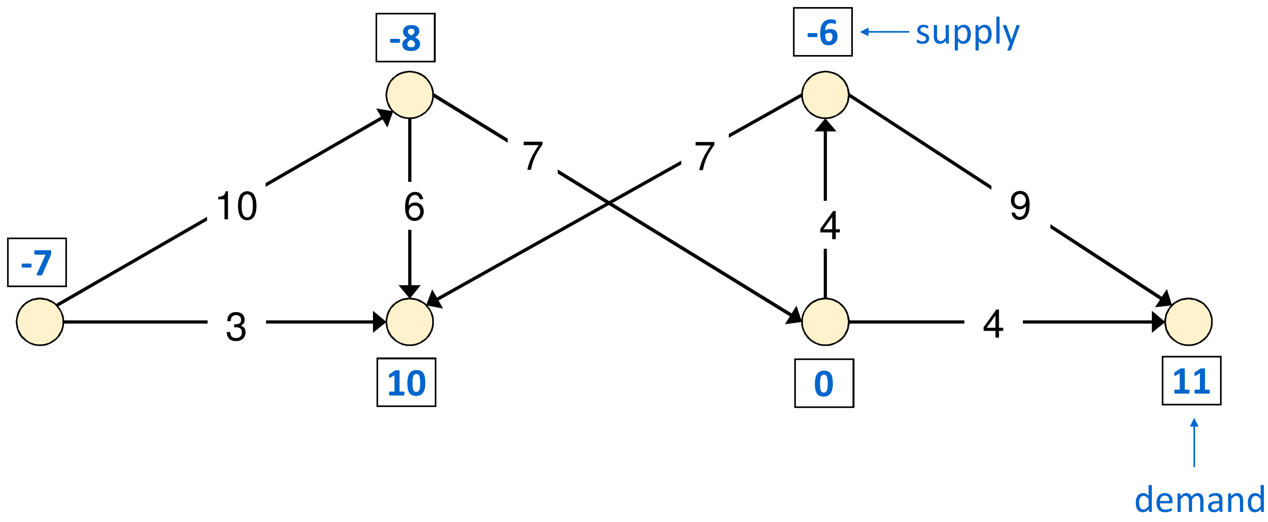
Defn: A circulation for (G, c, d) is a flow function $f: E \to \mathbb{R}$ meeting all the capacities, $0 \leq f(e) \leq c(e)$, and demands:
$\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$.

Circulation with Demands: Given (G, c, d), does it have a circulation? If so, find it.

Circulation with Demands

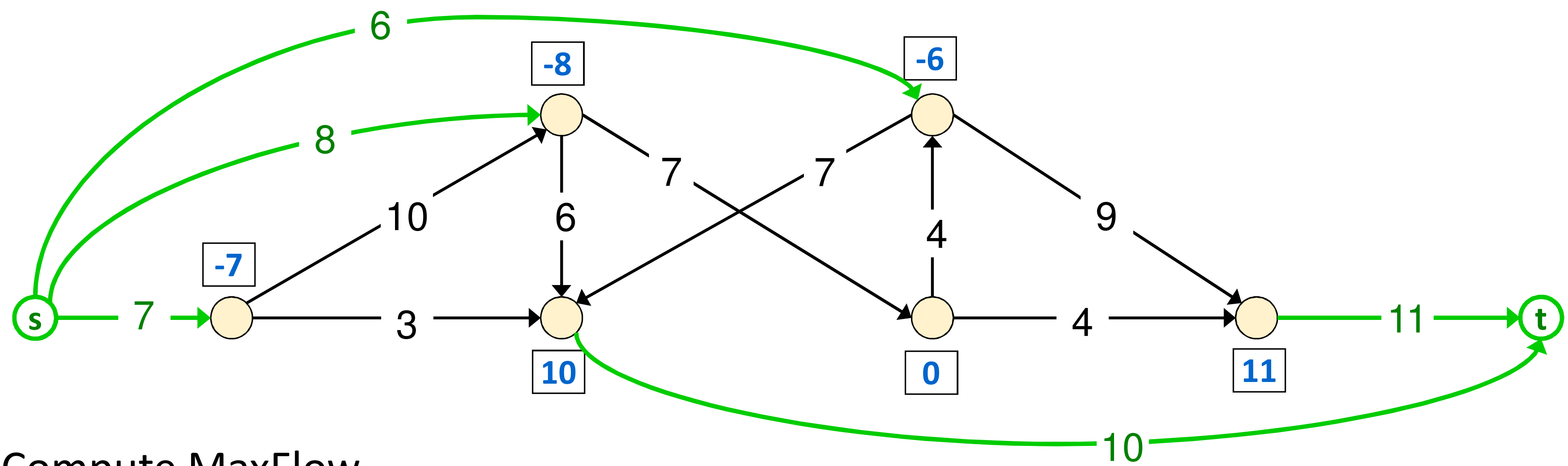
Defn: Total supply $D = \sum_{v:\, d(v)<0} |d(v)| = -\sum_{v:\, d(v)<0} d(v)$.

Necessary condition: $\sum_{v:\, d(v)>0} d(v) = D$ (no supply is lost)



Circulation with Demands using Network Flow

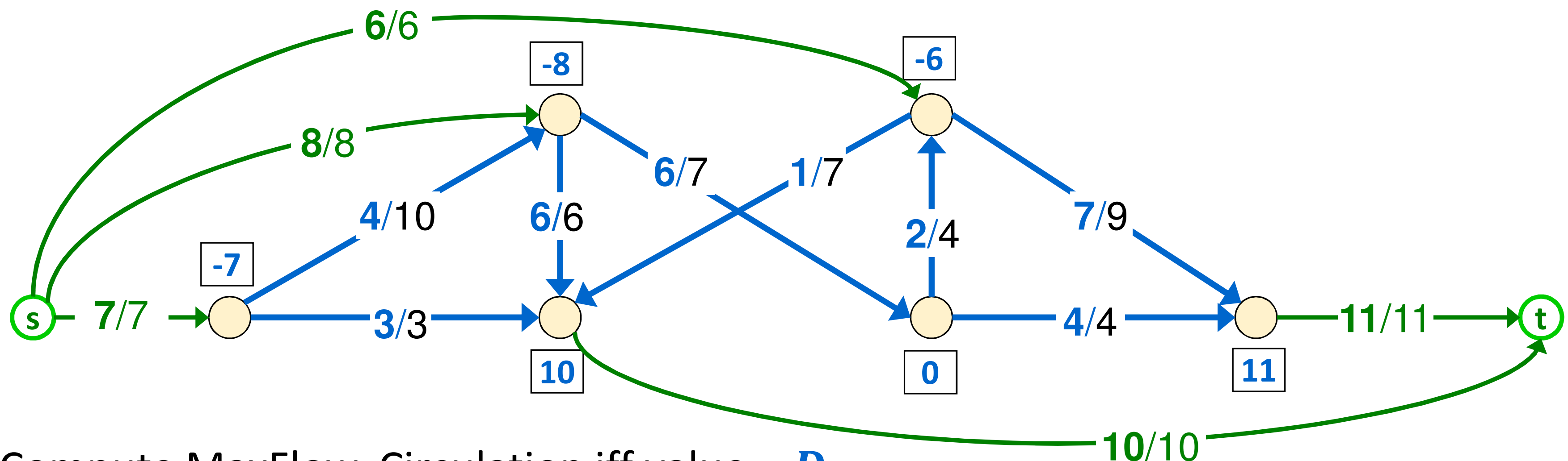
- Add new source s and sink t.
- Add edge (s, v) for all supply nodes v with capacity $|d(v)|$.
- Add edge (v, t) for all demand nodes v with capacity $d(v)$.



- Compute MaxFlow.

Circulation with Demands using Network Flow

- MaxFlow $\leq D$ based on cuts out of s or into t.
- If MaxFlow $= D$ then all supply/demands satisfied.

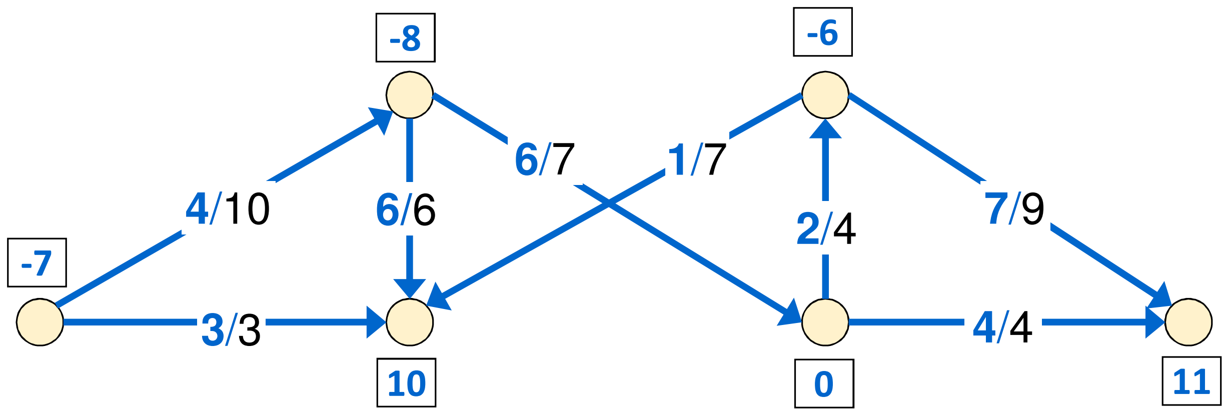


- Compute MaxFlow. Circulation iff value = D

Circulation with Demands using Network Flow

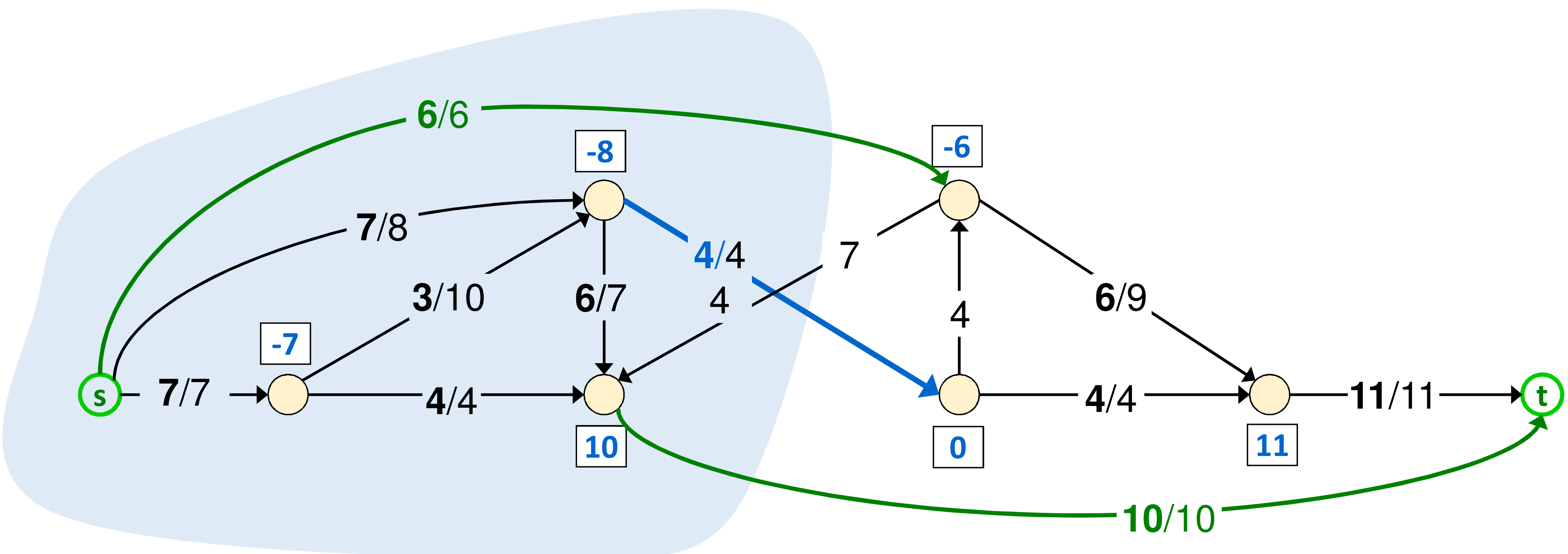
Circulation = flow on original edges

Circulations only need integer flows



Circulation with Demands using Network Flow

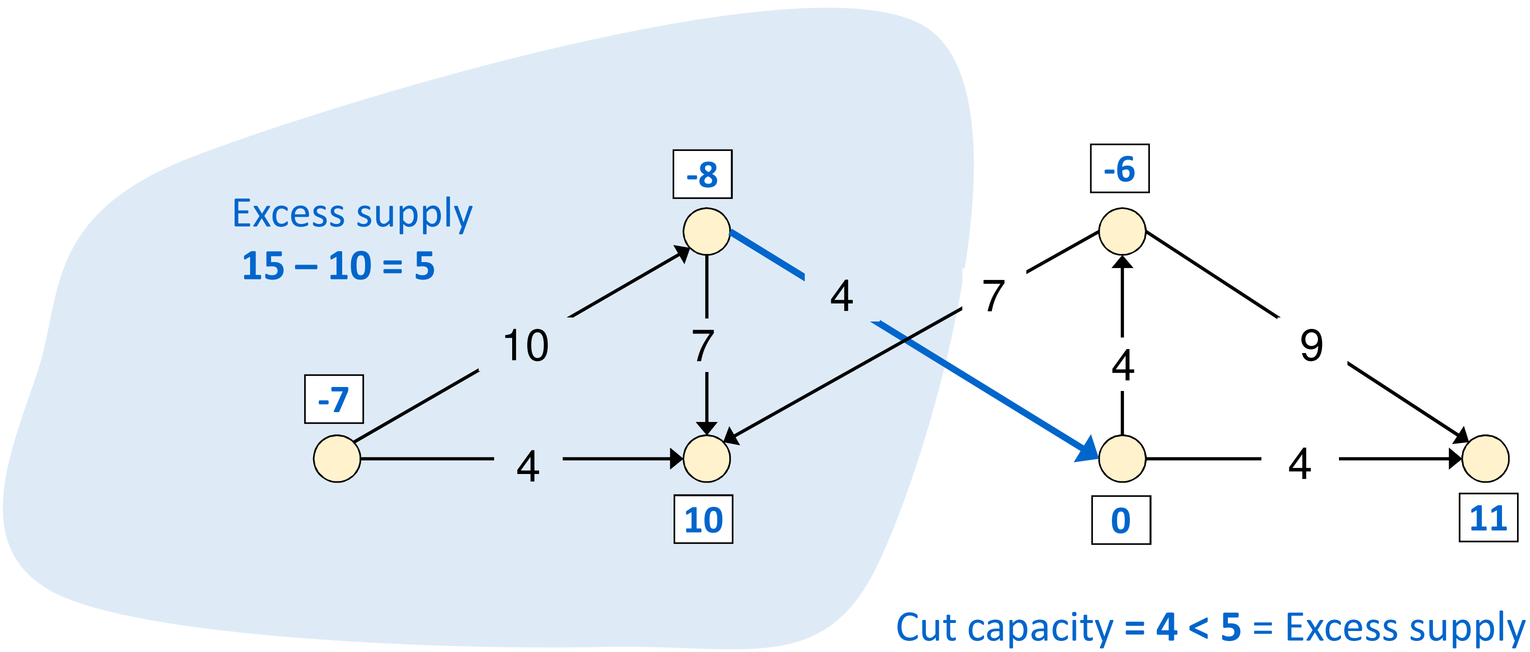
When does a circulation not exist? MaxFlow $< D$ iff MinCut $< D$.



Circulation with Demands using Network Flow

When does a circulation not exist? MaxFlow $< D$ iff MinCut $< D$.

Equivalent to excess supply on "source" side of cut smaller than cut capacity.



Some general ideas for using MaxFlow/MinCut

- If no source/sink, add them with appropriate capacity depending on application
- Sometimes can have edges with no capacity limits
  - Infinite capacity (or, equivalently, very large integer capacity)
- Convert undirected graphs to directed ones
- Can remove unnecessary flow cycles in answers
- Another idea:
  - To use them for vertex capacities c_v
    - Make two copies of each vertex v named v_{in}, v_{out}