CSC373

Weeks 5,6: Network Flow

Nisarg Shah

Recap

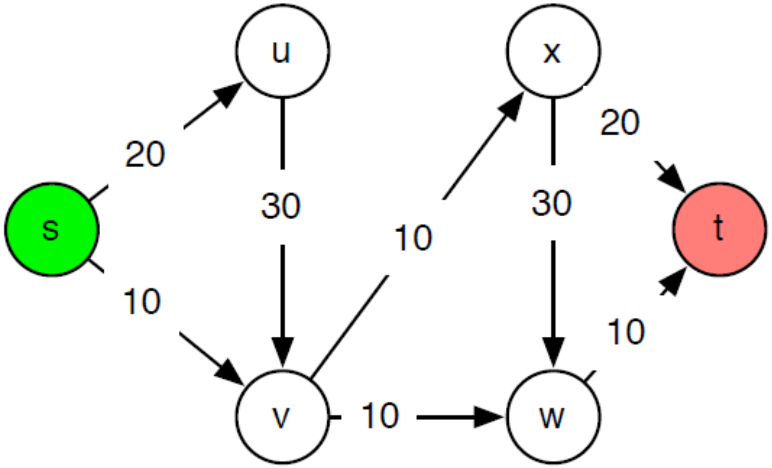
- Dynamic Programming Basics
  - Optimal substructure property
  - Bellman equation
  - Top-down (memoization) vs bottom-up implementations

- Dynamic Programming Examples
  - Weighted interval scheduling
  - Knapsack problem
  - Single-source shortest paths
  - Chain matrix product
  - Edit distance (aka sequence alignment)
  - Traveling salesman problem (TSP)

Network Flow

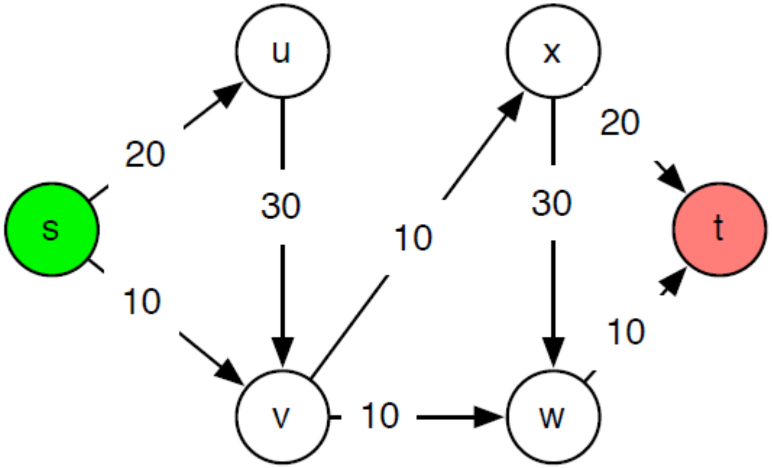
Network Flow

- Input
  - A directed graph $G = (V, E)$
  - Edge capacities $c : E \to \mathbb{R}_{\geq 0}$
  - Source node s, target node t
- Output
  - Maximum "flow" from s to t



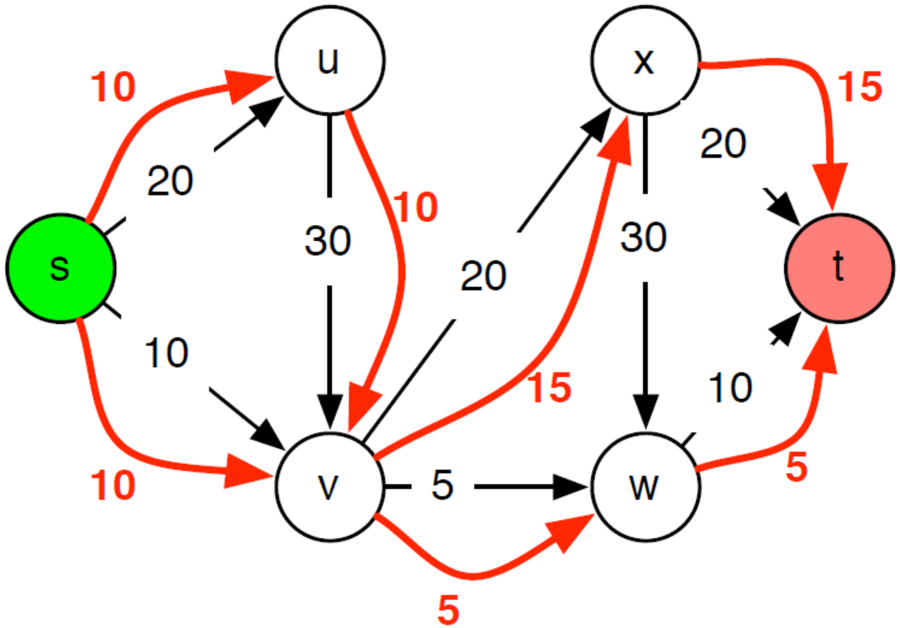
Network Flow

- Assumptions
  - No edges enter s
  - No edges leave t
  - Edge capacity $c(e)$ is a non-negative integer
    - Later, we'll see what happens when $c(e)$ can be a rational or irrational number



Network Flow

- Flow
  - An s-t flow is a function $f: E \rightarrow \mathbb{R}_{\geq 0}$
  - Intuitively, $f(e)$ is the "amount of material" carried on edge e



Network Flow

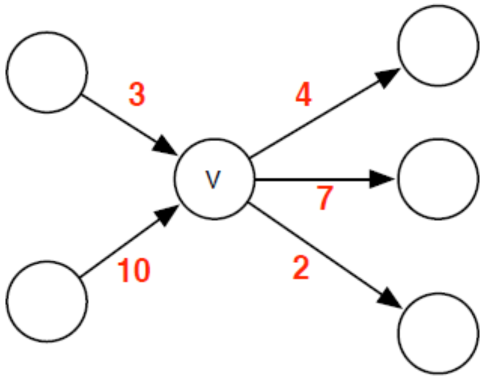
- Constraints on flow f

  1. Respecting capacities

     $\forall e \in E : 0 \leq f(e) \leq c(e)$

  2. Flow conservation

     $\forall v \in V \setminus \{s,t\} : \sum_{e \text{ entering } v} f(e) = \sum_{e \text{ leaving } v} f(e)$



Flow in = flow out at every node other than s and t

Network Flow

- $f^{in}(v) = \sum_{e \text{ entering } v} f(e)$
- $f^{out}(v) = \sum_{e \text{ leaving } v} f(e)$

- Value of flow f is $v(f) = f^{out}(s) = f^{in}(t)$
  - Q: Why is $f^{out}(s) = f^{in}(t)$?

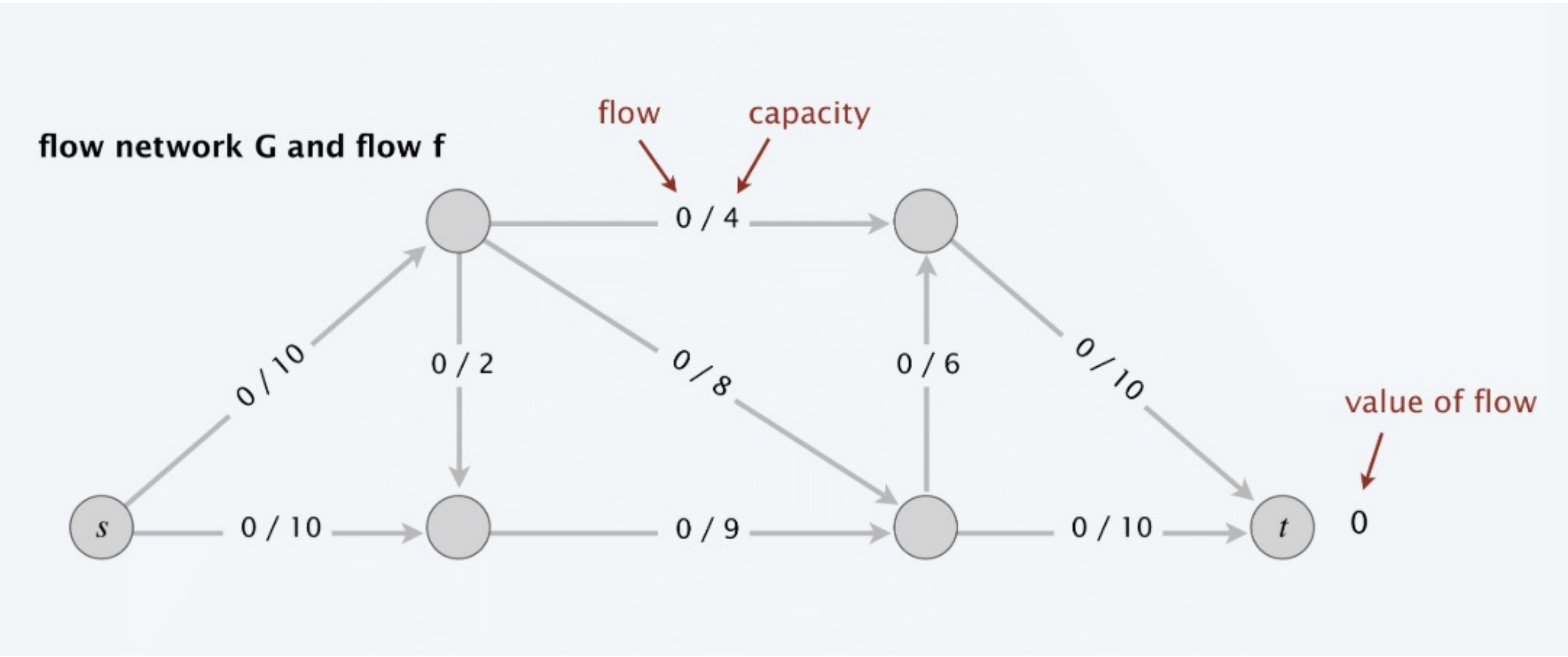
- Restating the problem:
  - Given a directed graph $G = (V, E)$ with edge capacities $c: E \to \mathbb{R}_{\geq 0}$, find a flow f^* with the maximum value.

First Attempt

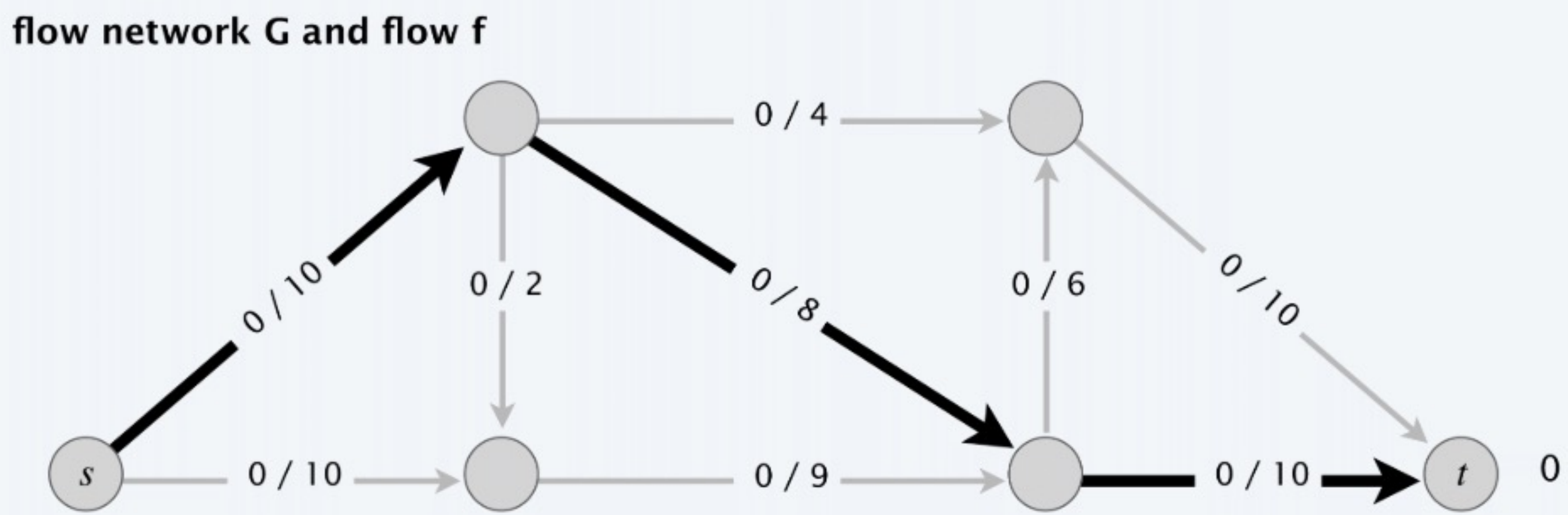
- A natural greedy approach
    1. Start from zero flow ($f(e) = 0$ for each e).
    2. While there exists an s-t path P in G such that $f(e) < c(e)$ for each $e \in P$
        a. Find any such path P
        b. Compute $\Delta = \min_{e \in P} \left(c(e) - f(e) \right)$
        c. Increase the flow on each edge $e \in P$ by Δ

- Note
    - Capacity and flow conservation constraints remain satisfied

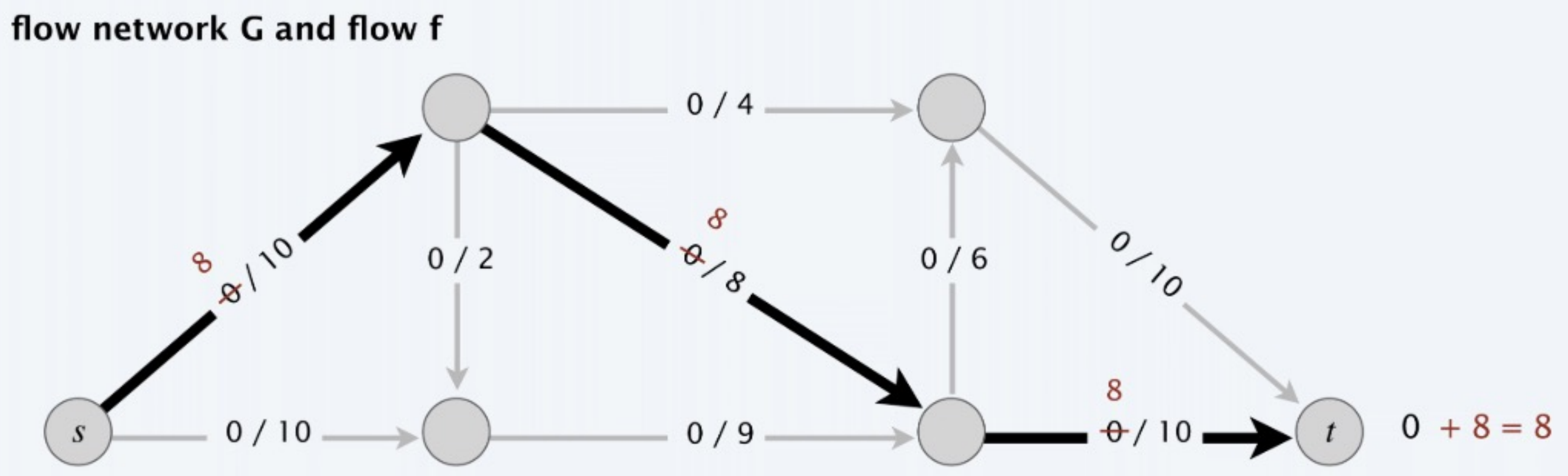
First Attempt



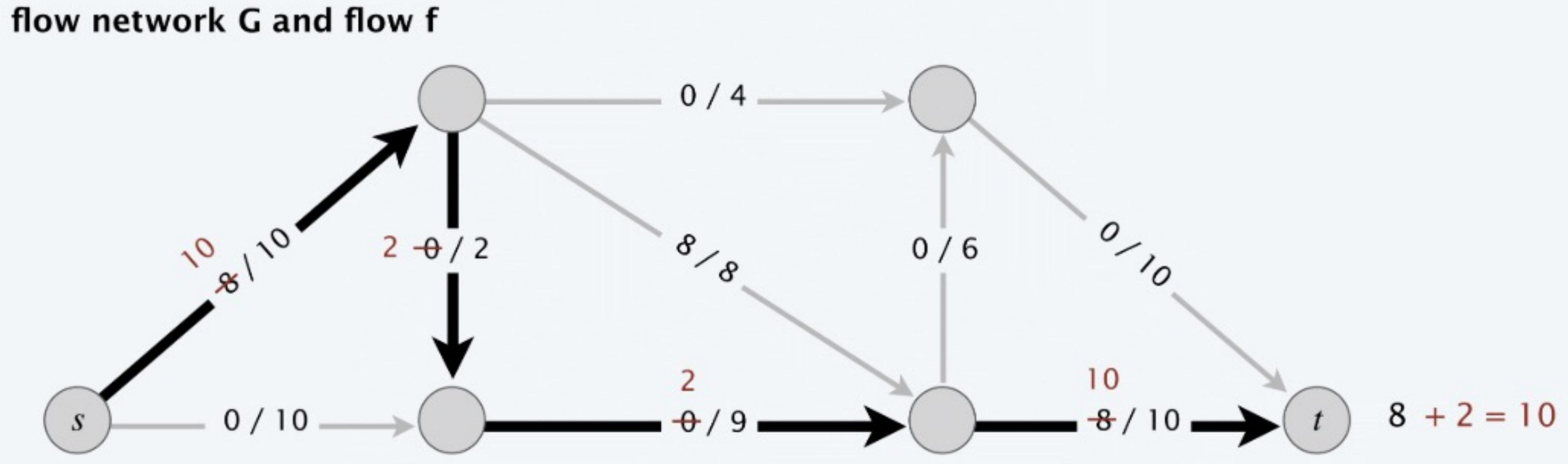
First Attempt



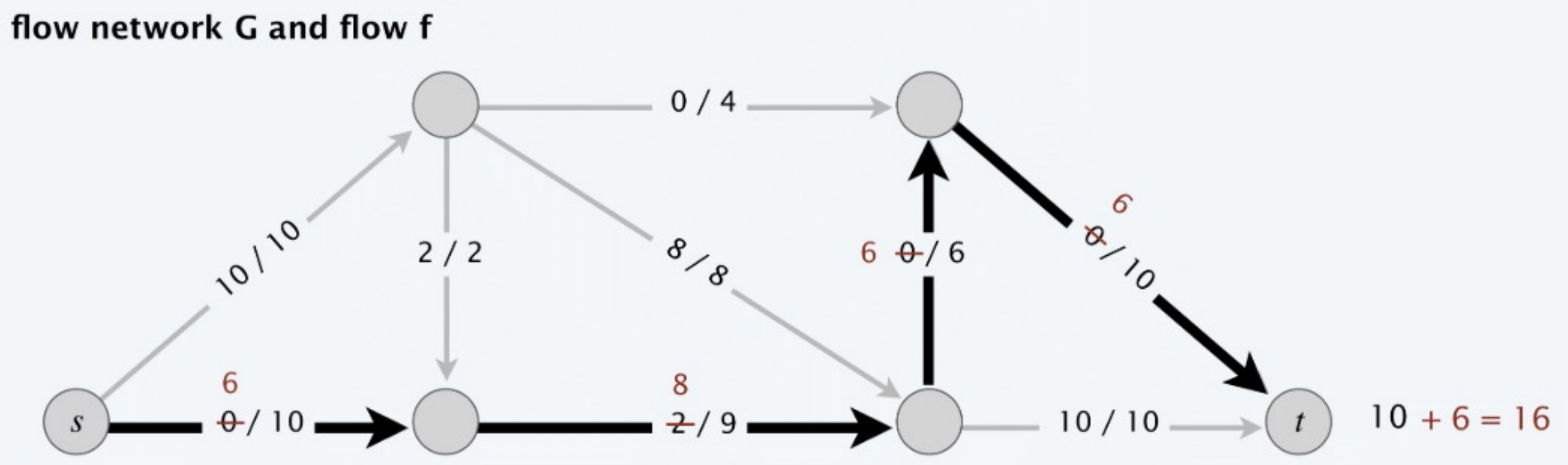
First Attempt



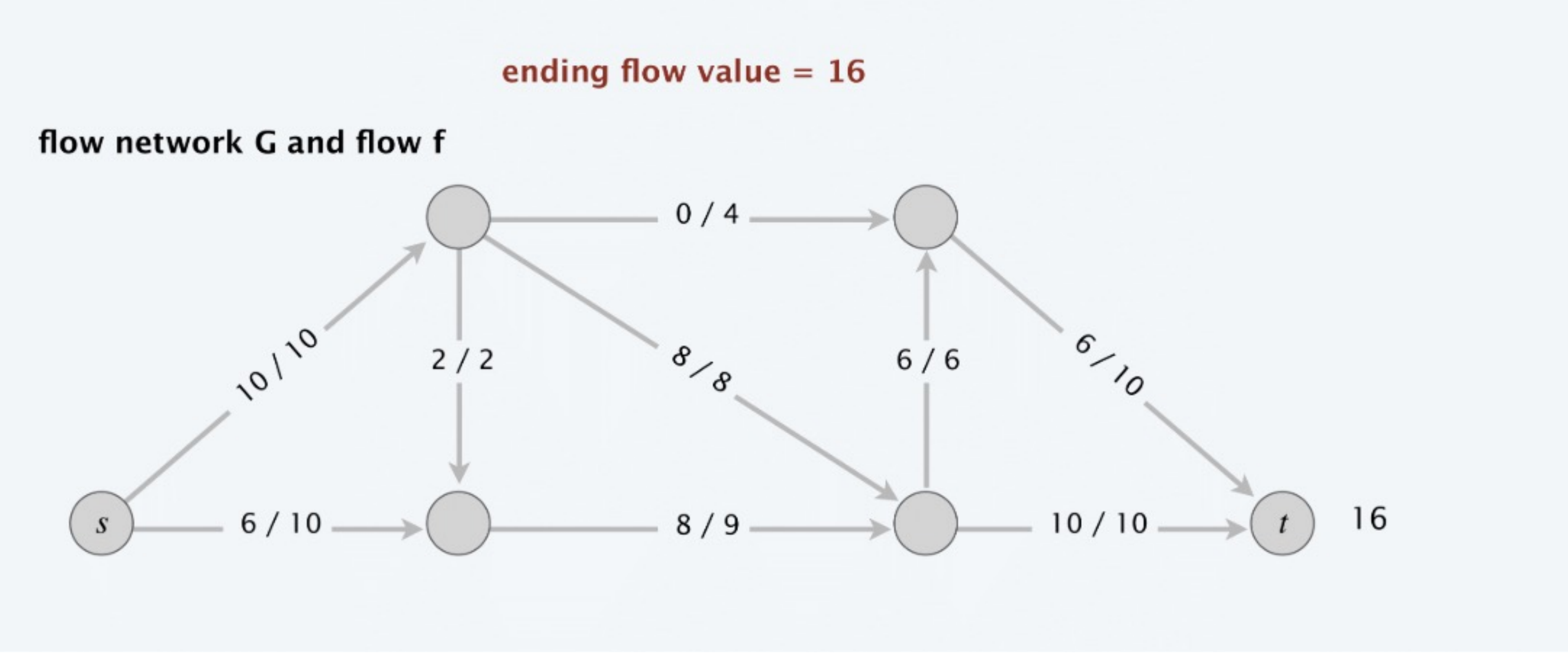
First Attempt



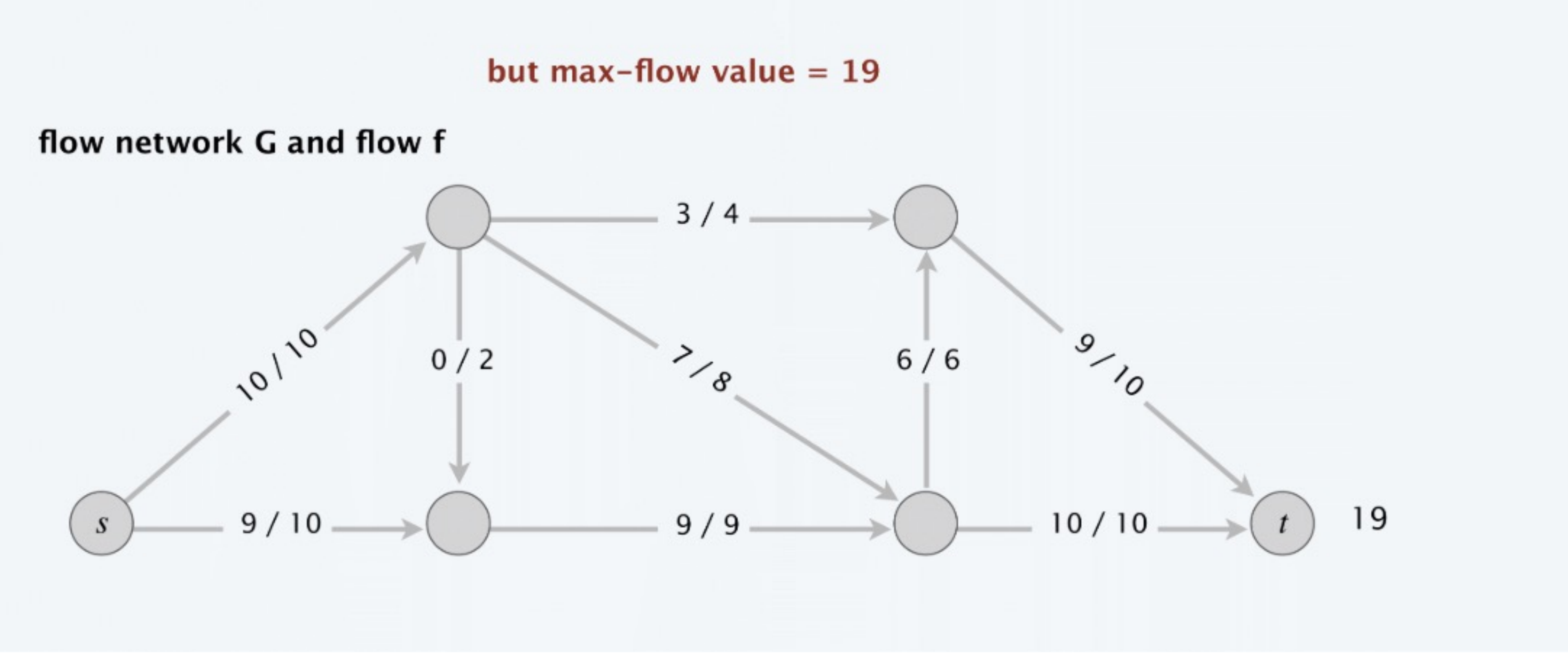
First Attempt



First Attempt



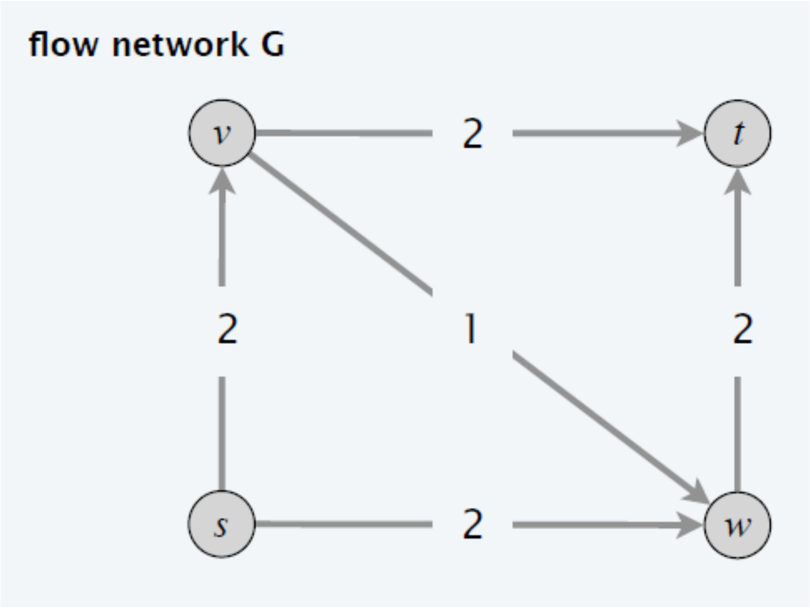
First Attempt



First Attempt

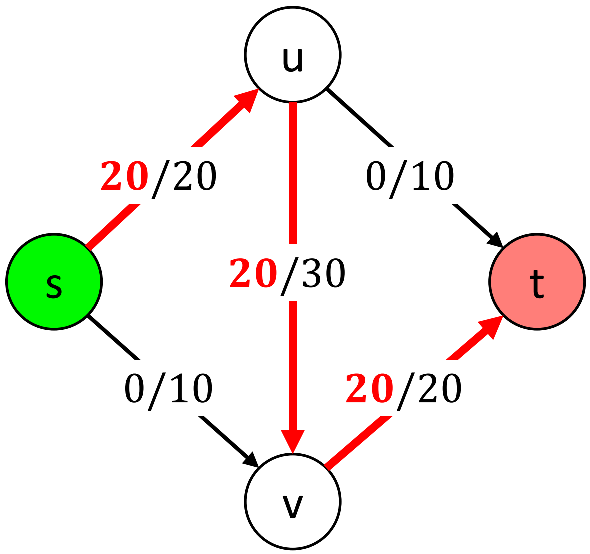
- Q: Why does the simple greedy approach fail?
- A: Because once it increases the flow on an edge, it is not allowed to decrease it ever in the future.

- Need a way to "reverse" bad decisions

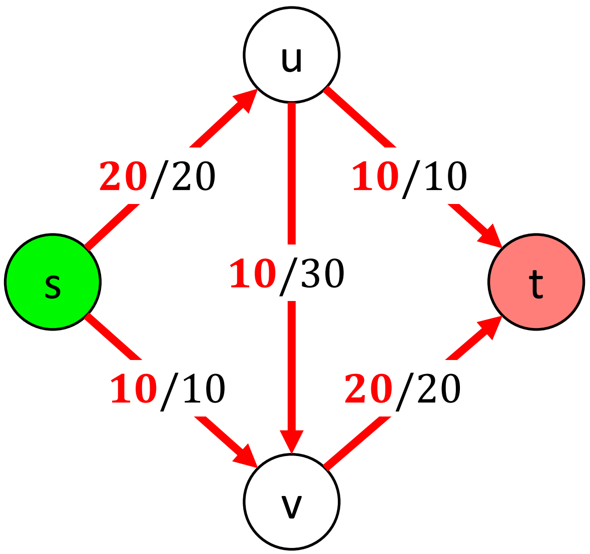


Reversing Bad Decisions

Suppose we start by sending 20 units of flow along this path

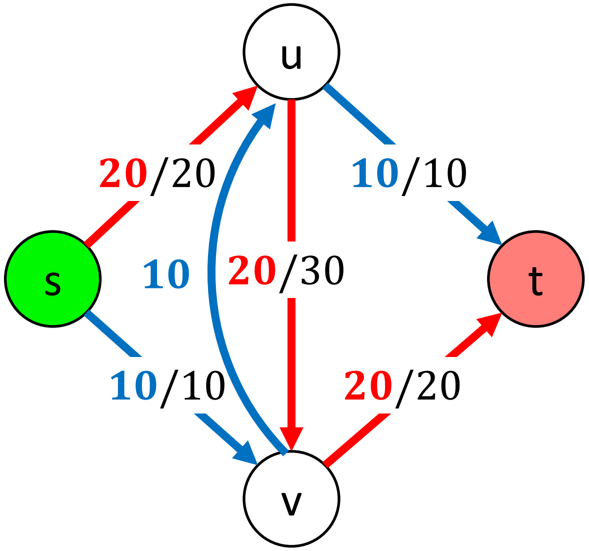


But the optimal configuration requires 10 fewer units of flow on $u \to v$

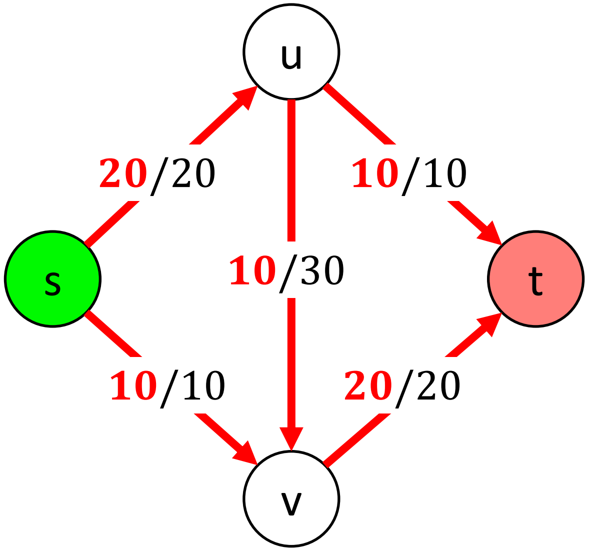


Reversing Bad Decisions

We can essentially send a "reverse" flow of 10 units along $v \rightarrow u$



So now we get this optimal flow



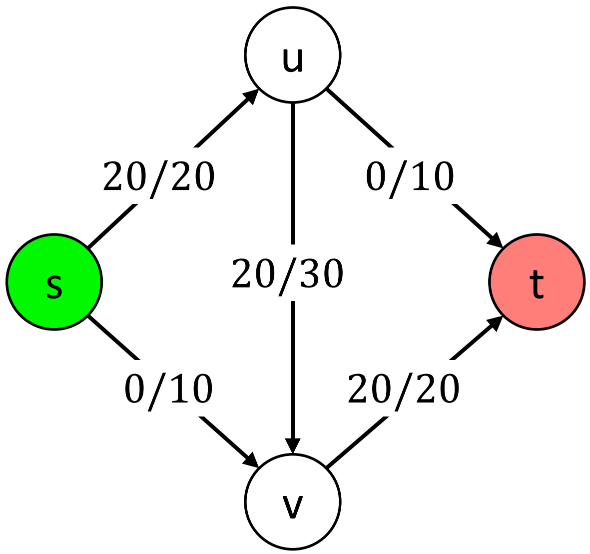
Residual Graph

- Suppose the current flow is f
- Define the residual graph G_f of flow f
  - G_f has the same vertices as G
  - For each edge e $= (u, v)$ in G, G_f has at most two edges
    - Forward edge $e = (u, v)$ with capacity $c(e) - f(e)$
      - We can send this much additional flow on e
    - Reverse edge $e^{rev} = (v, u)$ with capacity $f(e)$
      - The maximum "reverse" flow we can send is the maximum amount by which we can reduce flow on e, which is $f(e)$
    - We only really add edges of capacity > 0

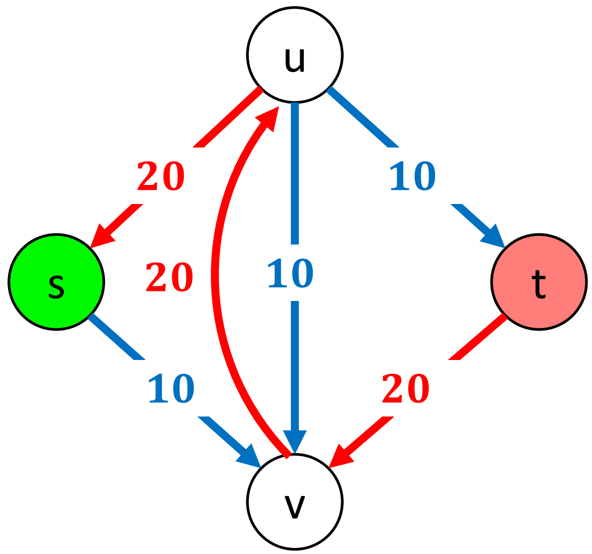
Residual Graph

- Example!

Flow f



Residual graph G_f



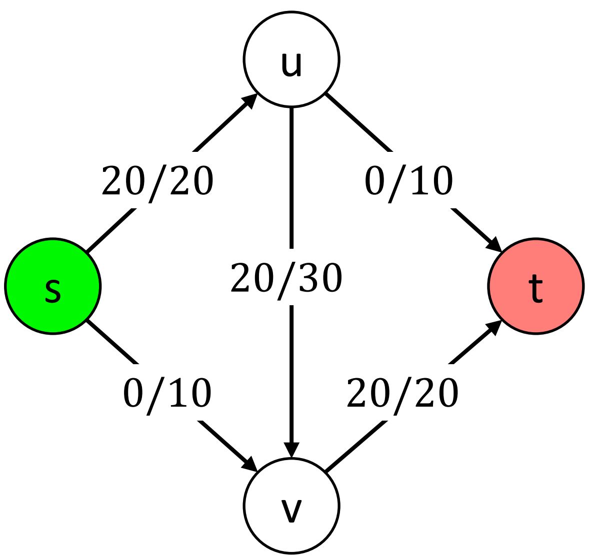
Augmenting Paths

- Let P be an s-t path in the residual graph G_f
- Let $\text{bottleneck}(P, f)$ be the smallest capacity across all edges in P
- "Augment" flow f by "sending" $\text{bottleneck}(P, f)$ units of flow along P
  - What does it mean to send x units of flow along P?
  - For each forward edge $e \in P$, increase the flow on e by x
  - For each reverse edge $e^{rev} \in P$, decrease the flow on e by x

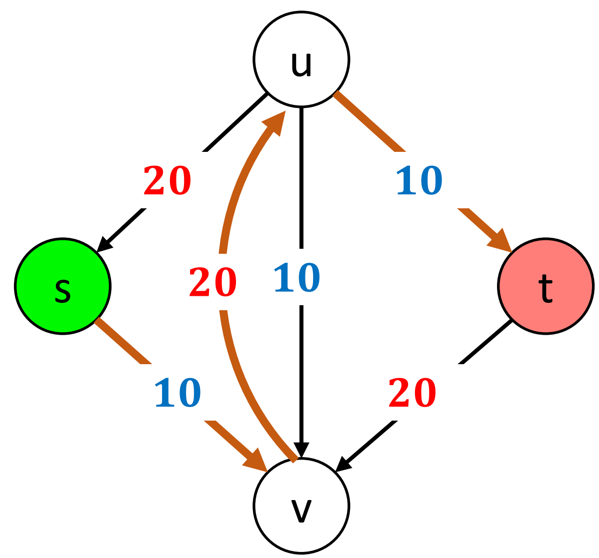
Residual Graph

- Example!

Flow f



Residual graph G_f

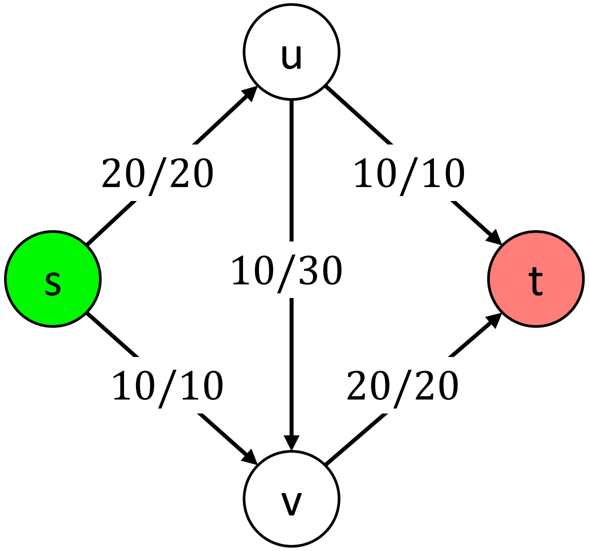


Path P → send flow = bottleneck = 10

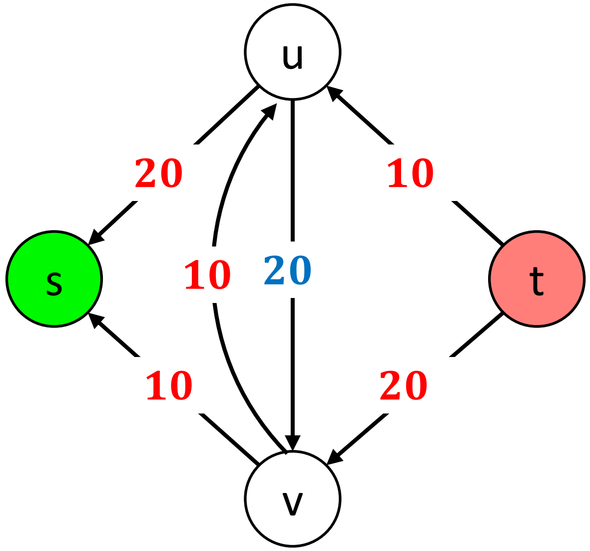
Residual Graph

- Example!

New flow f



New residual graph G_f



No s-t path because no outgoing edge from s

Augmenting Paths

- Let's argue that the new flow is a valid flow

- Capacity constraints (easy):

  - If we increase flow on e, we can do so by at most the capacity of forward edge e in G_f, which is $c(e) - f(e)$
    - So, the new flow can be at most $f(e) + \big(c(e) - f(e)\big) = c(e)$

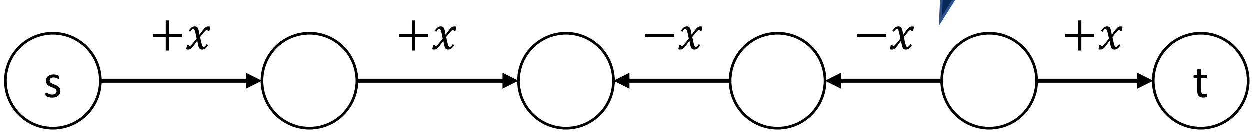
  - If we decrease flow on e, we can do so by at most the capacity of reverse edge e^{rev} in G_f, which is $f(e)$
    - So, the new flow is at least $f(e) - f(e) = 0$

Augmenting Paths

- Let's argue that the new flow is a valid flow

- Flow conservation (a bit trickier):
  - Each node on the path (except s and t) has exactly two incident edges
    - Both forward / both reverse ⇒ one is incoming, one is outgoing
      - Flow increased on both or decreased on both
    - One forward, one reverse ⇒ both incoming / both outgoing
      - Flow increased on one but decreased on the other
    - In each case, net flow remains 0

Edge directions as in G



Ford-Fulkerson Algorithm

```
MaxFlow(G):

  // initialize:
  Set f(e) = 0 for all e in G

  // while there is an s-t path in G_f:
  While P = FindPath(s,t,Residual(G,f))!=None:
    f = Augment(f,P)
    UpdateResidual(G,f)
  EndWhile

  Return f
```

Ford-Fulkerson Algorithm

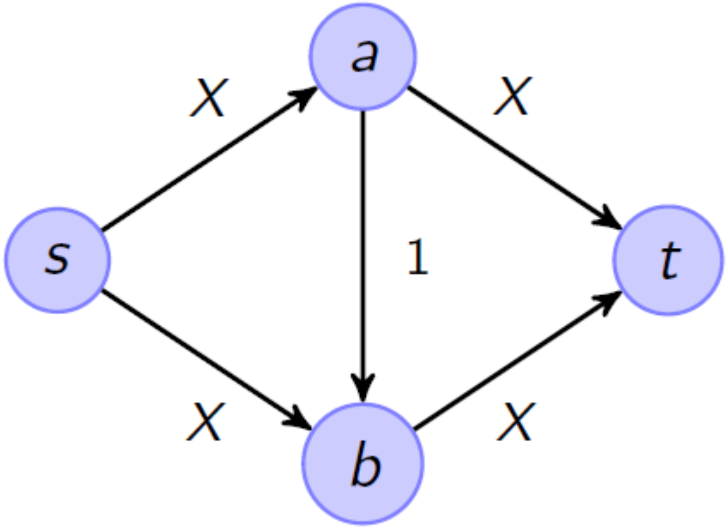
- Running time:
  - #Augmentations:
    - At every step, flow and capacities remain integers
    - For path P in G_f, $\text{bottleneck}(P, f) > 0$ implies $\text{bottleneck}(P, f) \geq 1$
    - Each augmentation increases flow by at least 1
    - Max flow (hence max #augmentations) is at most $C = \sum_{e \text{ leaving } s} c(e)$
  - Time to perform an augmentation:
    - G_f has n vertices and at most $2m$ edges
    - Finding P, computing $\text{bottleneck}(P, f)$, updating G_f
      - $O(m + n)$ time
  - Total time: $O((m + n) \cdot C)$

Ford-Fulkerson Algorithm

- Total time: $O((m+n) \cdot C)$
  - This is NOT polynomial time
  - The value of C can be exponentially large in the input length (the number of bits required to write down the edge capacities)
  - Note: While we assumed integer capacities, we know that the algorithm must always terminate even with rational capacities.
    - Why?
    - With irrational capacities, there is an example in which the algorithm never terminates.
- Q: Can we convert this to polynomial time?

Ford-Fulkerson Algorithm

- Q: Can we convert this to polynomial time?
  - Not if we choose an *arbitrary* path in G_f at each step
  - In the graph below, we might end up repeatedly sending 1 unit of flow across $a \to b$ and then reversing it
    - Takes X steps, which can be exponential in the input length



Ford-Fulkerson Algorithm

- Ways to achieve polynomial time
  - Find the maximum bottleneck capacity augmenting path
    - Runs in $O(m^2 \cdot \log C)$ operations
  - Find the shortest augmenting path using BFS
    - Edmonds-Karp algorithm
    - Runs in $O(nm^2)$ operations
    - Can be found in CLRS
  - …

Max Flow Problem

- Race to reduce the running time
  - 1972: $O(n\,m^2)$ Edmonds-Karp
  - 1980: $O(n\,m\log^2 n)$ Galil-Namaad
  - 1983: $O(n\,m\log n)$ Sleator-Tarjan
  - 1986: $O\left(n\,m\log\left({}^{n^2}\!/_{m}\right)\right)$ Goldberg-Tarjan
  - 1992: $O(n\,m + n^{2+\epsilon})$ King-Rao-Tarjan
  - 1996: $O\left(n\,m\frac{\log n}{\log {}^{m}\!/_{n\log n}}\right)$ King-Rao-Tarjan
    - Note: These are $O(n\,m)$ when $m = \omega(n)$
  - 2013: $O(n\,m)$ Orlin
    - Breakthrough!
  - 2021: $O((m + n^{1.5}) \cdot \log X)$, where X = max edge capacity
    - Breakthrough based on very heavy techniques!

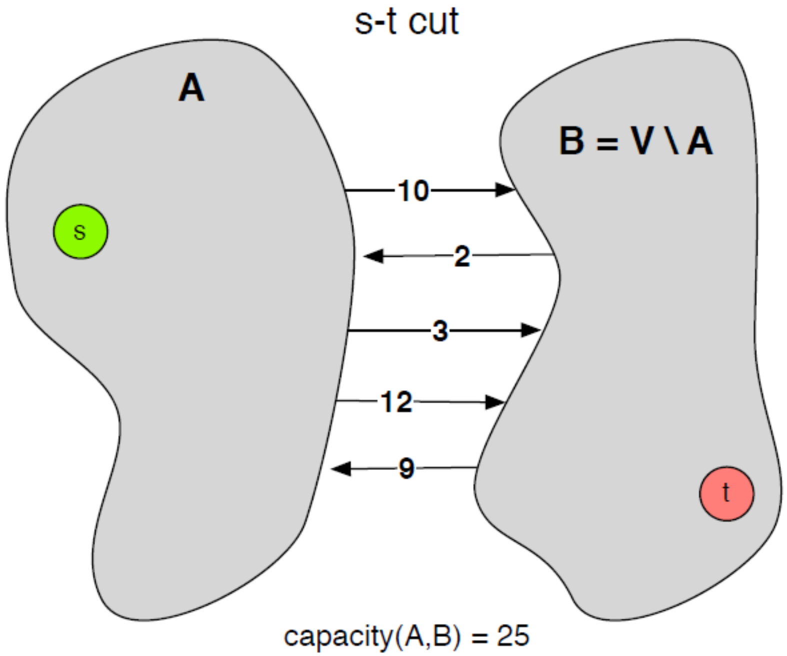
Back to Ford-Fulkerson

- We argued that the algorithm must terminate, and must terminate in $O\big((m+n)\cdot C\big)$ time

- But we didn't argue correctness yet, i.e., the algorithm must terminate with the optimal flow

Cuts and Cut Capacities

- (A, B) is an *s-t cut* if it is a partition of vertex set V (i.e., $A \cup B = V$, $A \cap B = \emptyset$) with $s \in A$ and $t \in B$
- Its capacity, denoted $cap(A, B)$, is the sum of capacities of edges *leaving* A

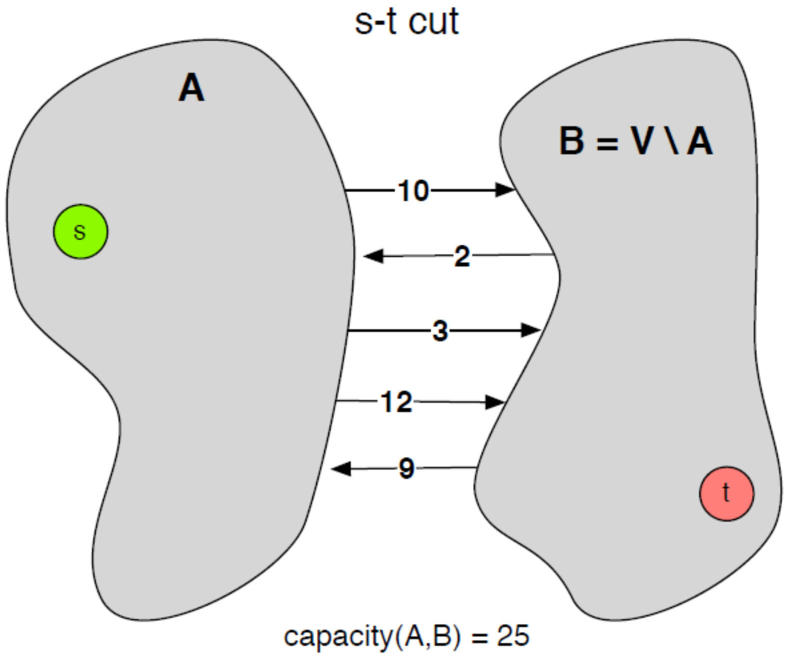


Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B),

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

- Proof (on the board): Just take a sum of the flow conservation constraint over all nodes in A



Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B),

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

- Proof:

$$
\begin{aligned}
v(f) &= f^{out}(A) - f^{in}(A) \\
&\leq f^{out}(A) \\
&= \textstyle\sum_{e \text{ leaving } A} f(e) \\
&\leq \textstyle\sum_{e \text{ leaving } A} c(e) \\
&= cap(A, B)
\end{aligned}
$$

Cuts and Flows

- Theorem: For any flow f and any s-t cut (A, B),

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

- Hence, $\max_f v(f) \leq \min_{(A,B)} cap(A, B)$

  - Max value of any flow $\leq$ min capacity of any s-t cut

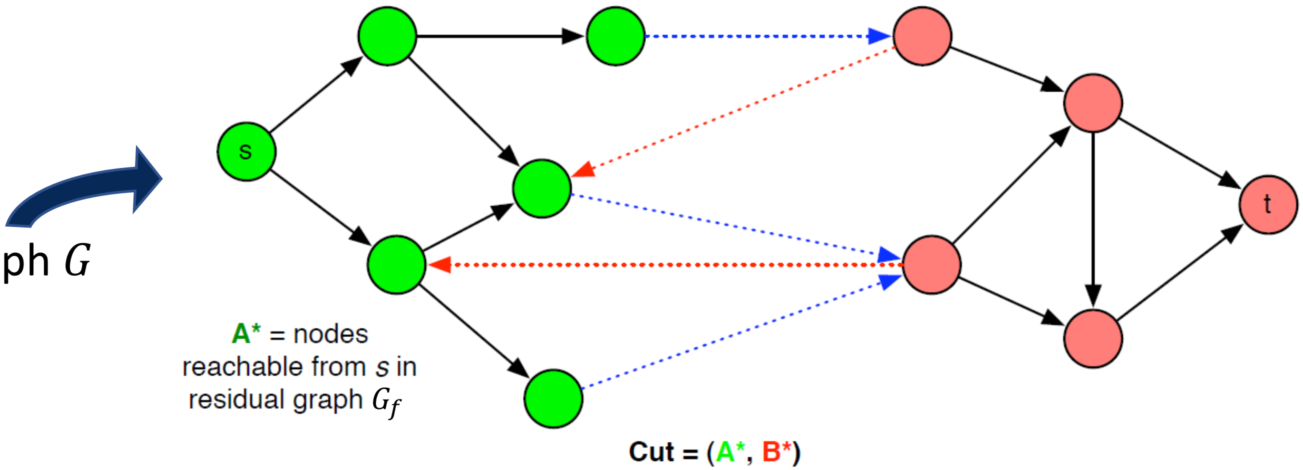
- We will now prove:
  - Value of flow generated by Ford-Fulkerson = capacity of *some* cut

- Implications
  - 1) Max flow = min cut
  - 2) Ford-Fulkerson generates max flow.

Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - f = flow returned by Ford-Fulkerson
  - A^* = nodes reachable from s in G_f
  - B^* = remaining nodes $V \setminus A^*$
  - Note: We look at the residual graph G_f, but define the cut in G

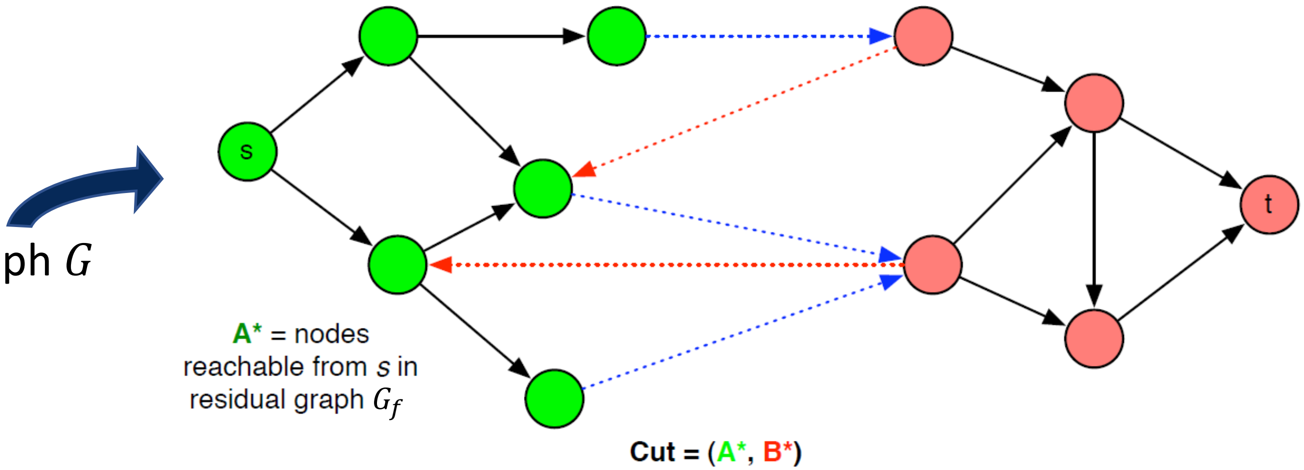
Graph G



Cuts and Flows

- **Theorem:** Ford-Fulkerson finds maximum flow.
- **Proof:**
  - Claim: (A^*, B^*) is a valid cut
    - $s \in A^*$ by definition
    - $t \in B^*$ because when Ford-Fulkerson terminates, there are no s-t paths in G_f, so $t \notin A^*$

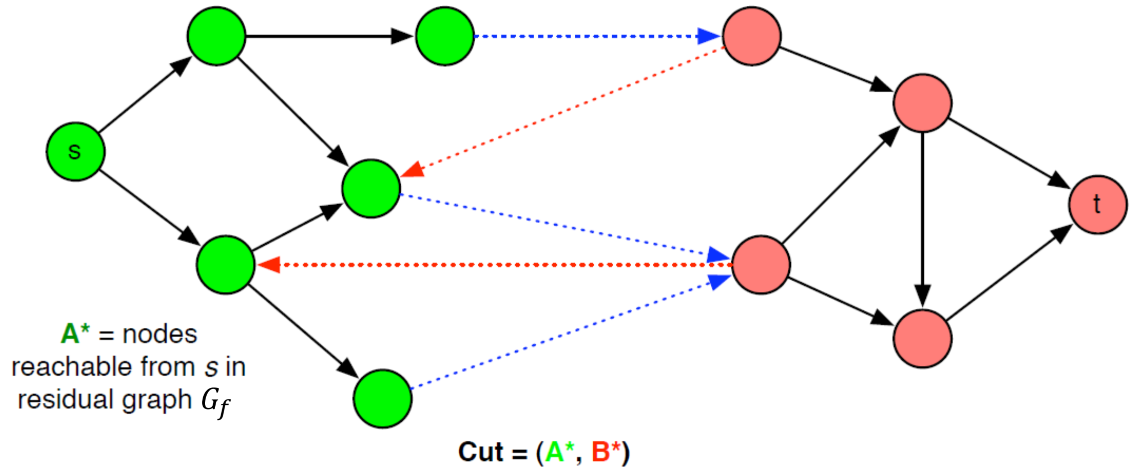
Graph G



Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Blue edges = edges going out of A^* $\boxed{\text{in } G}$
  - Red edges = edges coming into A^* $\boxed{\text{in } G}$

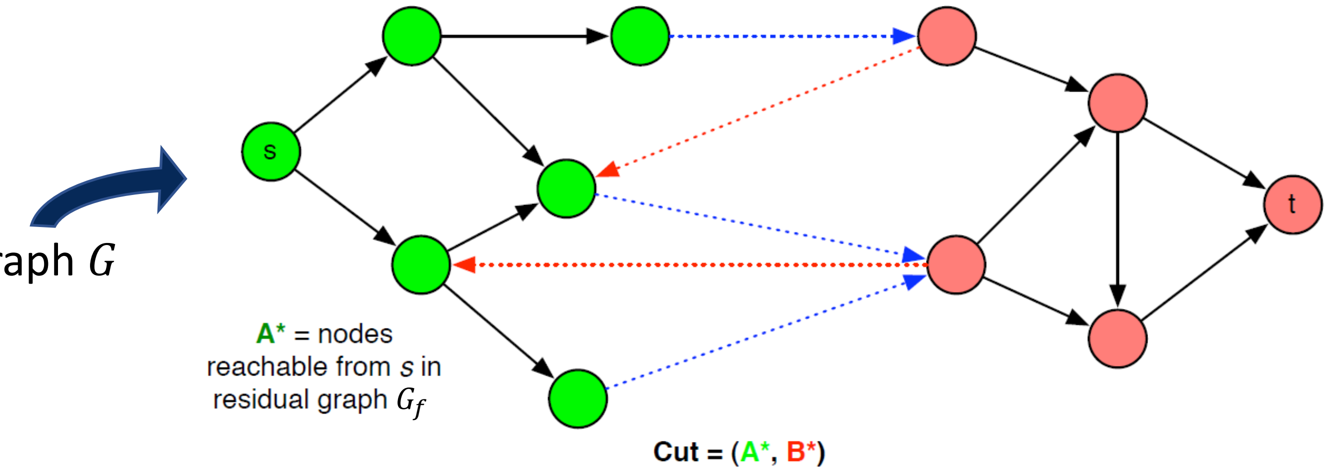
Graph G



Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Each blue edge (u, v) must be saturated
    - Otherwise G_f would have its forward edge (u, v) and then $v \in A^*$
  - Each red edge (v, u) must have zero flow
    - Otherwise G_f would have its reverse edge (u, v) and then $v \in A^*$

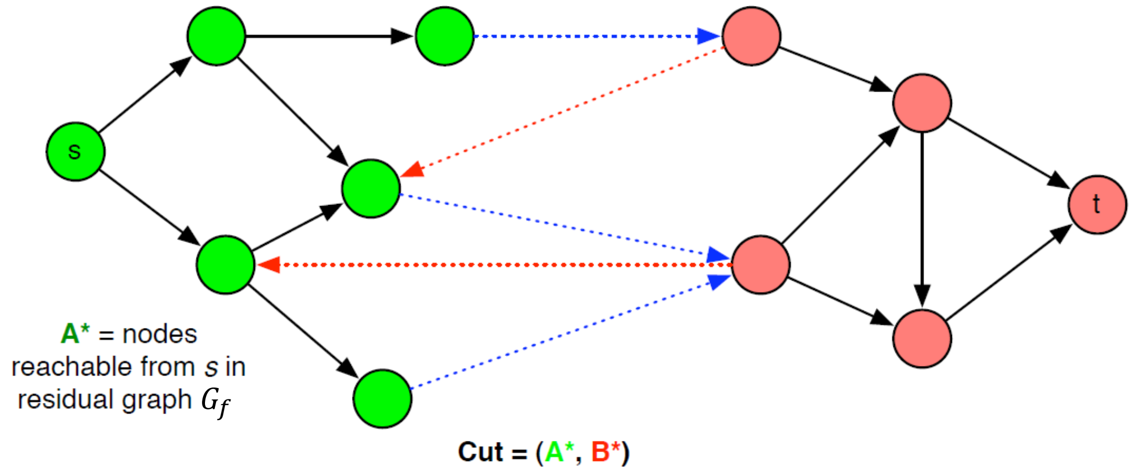
Graph G



Cuts and Flows

- Theorem: Ford-Fulkerson finds maximum flow.
- Proof:
  - Each blue edge (u, v) must be saturated $\Rightarrow f^{out}(A^*) = cap(A^*, B^*)$
  - Each red edge (v, u) must have zero flow $\Rightarrow f^{in}(A^*) = 0$
  - So $v(f) = f^{out}(A^*) - f^{in}(A^*) = cap(A^*, B^*)$ ■

Graph G



Max Flow - Min Cut

- Max Flow-Min Cut Theorem:
  In any graph, the value of the maximum flow is equal to the capacity of the minimum cut.

- Our proof already gives an algorithm to find a min cut
  - Run Ford-Fulkerson to find a max flow f
  - Construct its residual graph G_f
  - Let $A^* =$ set of all nodes reachable from s in G_f
    - Easy to compute using BFS
  - Then $(A^*, V \setminus A^*)$ is a min cut

Poll

Question

- There is a network G with positive integer edge capacities.
- You run Ford-Fulkerson.
- It finds an augmenting path with bottleneck capacity 1, and after that iteration, it terminates with a final flow value of 1.
- Which of the following statement(s) must be correct about G?

(a) G has a single s-t path.

(b) G has an edge e such that all s-t paths go through e.

(c) The minimum cut capacity in G is greater than 1.

(d) The minimum cut capacity in G is less than 1.

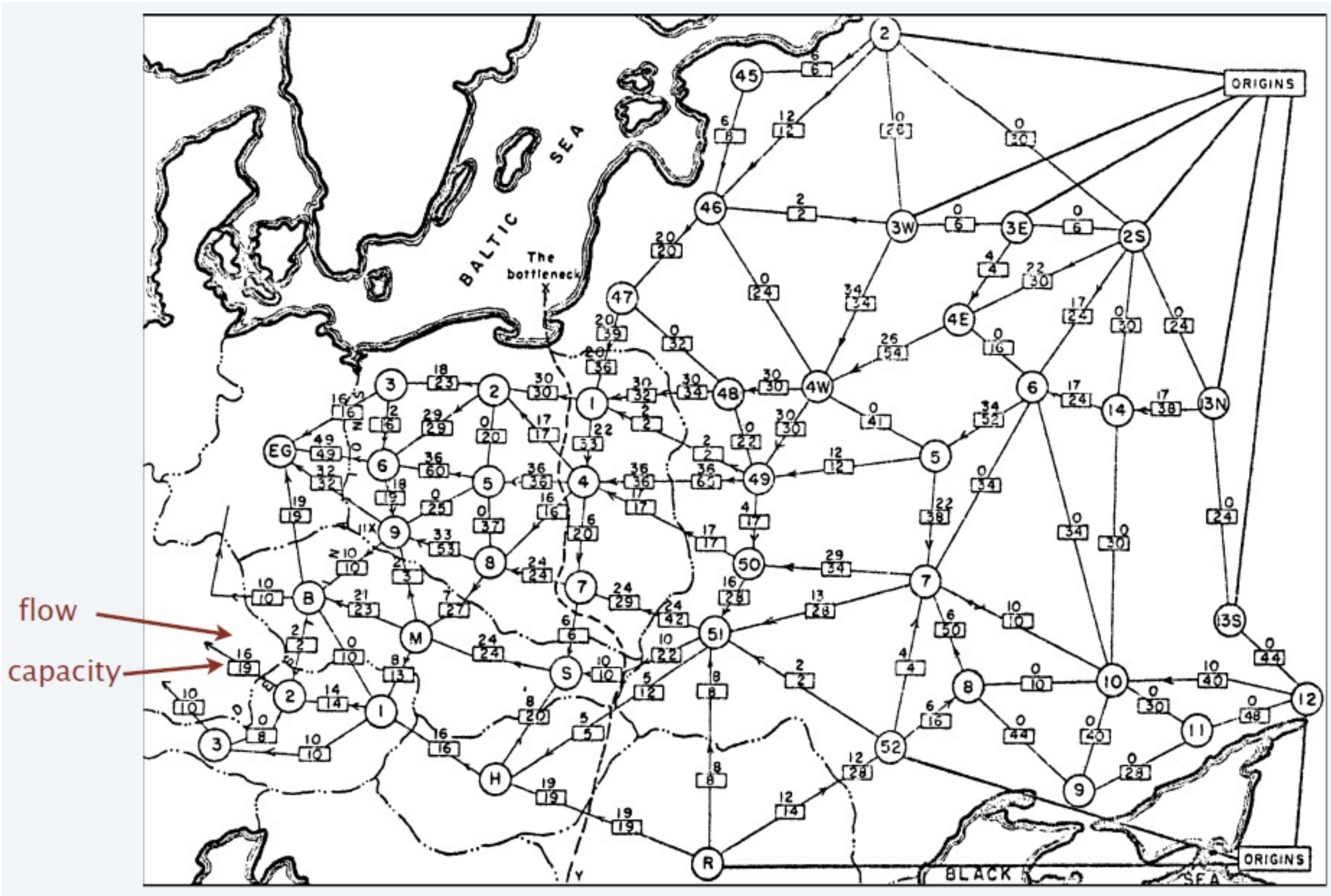
Why Study Flow Networks?

- Unlike divide-and-conquer, greedy, or DP, this doesn't seem like an algorithmic framework
  - It seems more like a single problem
- Turns out that many problems can be reduced to this versatile single problem
- Next lecture!

Network Flow Applications

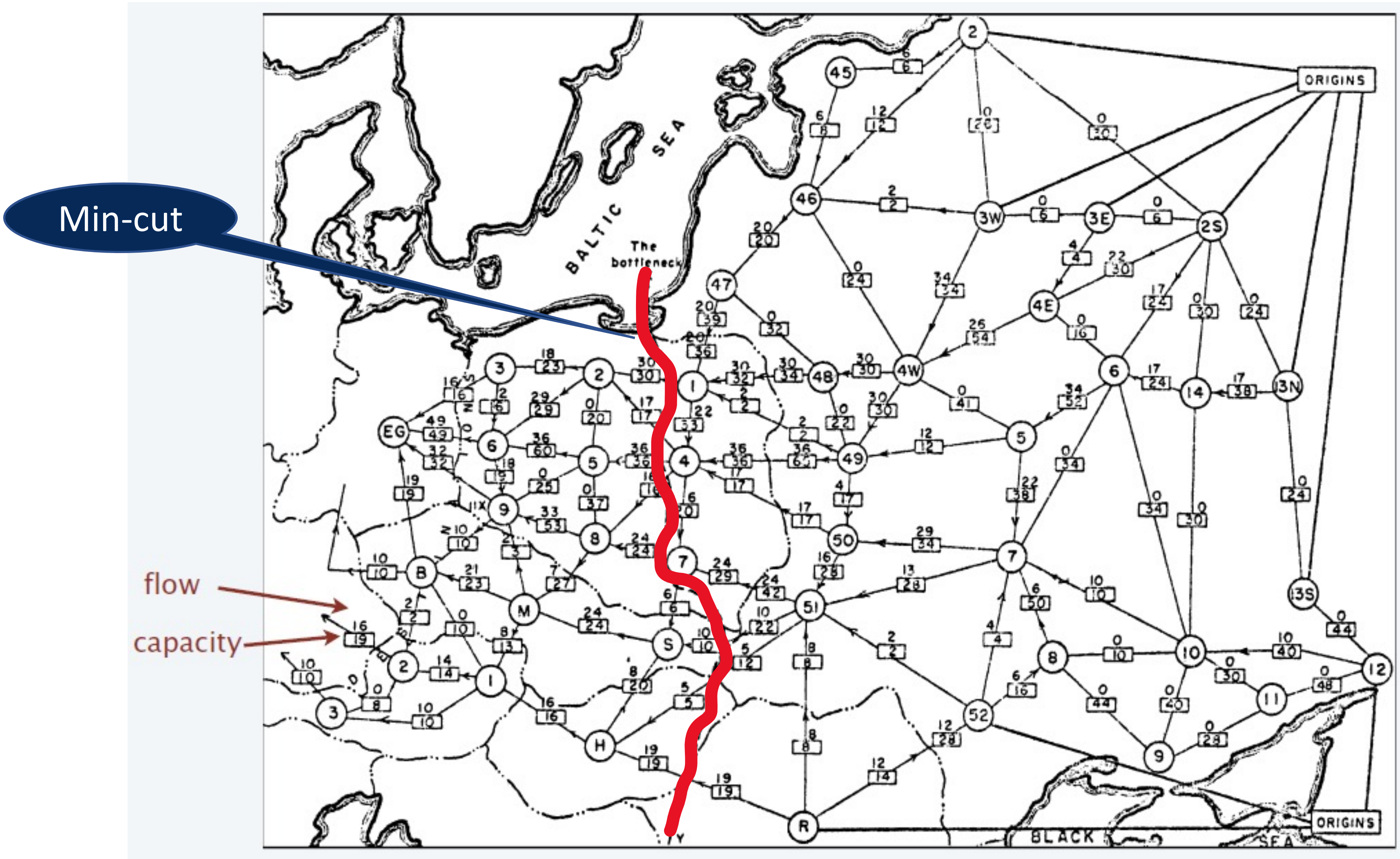
Rail network connecting Soviet Union with Eastern European countries

(Tolstoǐ 1930s)



Rail network connecting Soviet Union with Eastern European countries

(Tolstoǐ 1930s)



Integrality Theorem

- Before we look at applications, we need the following special property of the max-flow computed by Ford-Fulkerson and its variants

- Observation:
  - If edge capacities are integers, then the max-flow computed by Ford-Fulkerson and its variants are also integral (i.e., the flow on each edge is an integer).
  - Easy to check that each augmentation step preserves integral flow

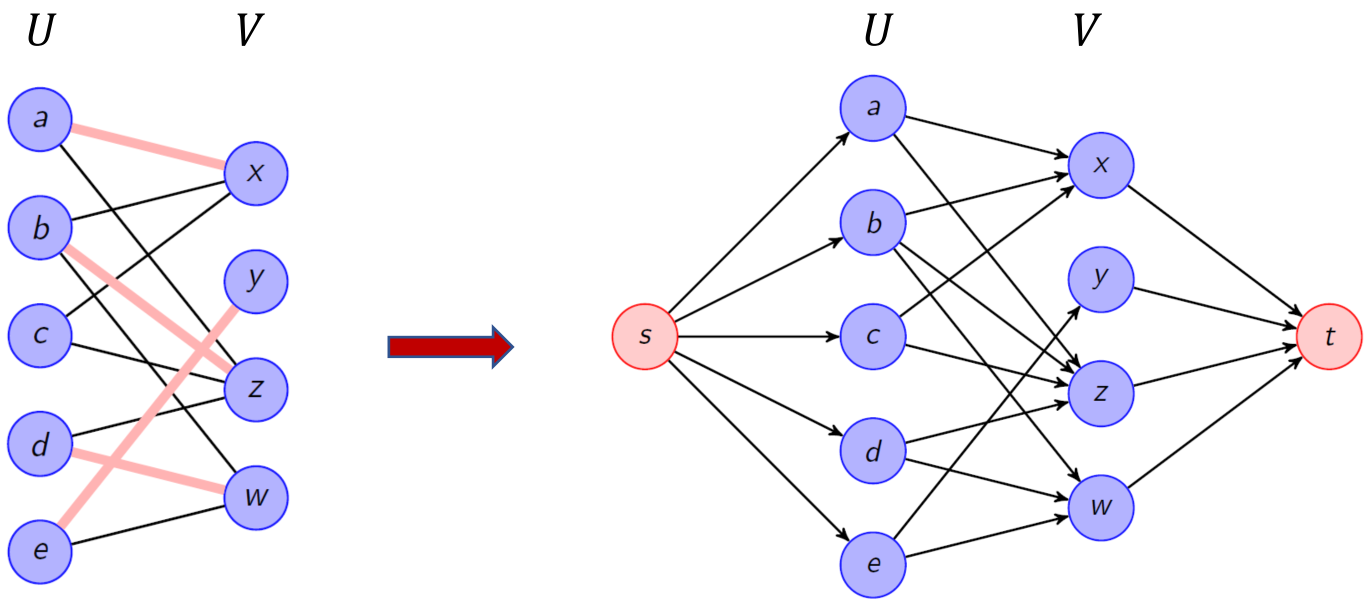
Bipartite Matching

- Problem
  - Given a bipartite graph $G = (U \cup V, E)$, find a maximum cardinality matching

- We do not know any efficient greedy or dynamic programming algorithm for this problem.

- But it can be reduced to max-flow.

Bipartite Matching



- Create a directed flow graph where we…
  - ➢ Add a source node s and target node t
  - ➢ Add edges, all of capacity 1:
    - ○ $s \to u$ for each $u \in U$, $v \to t$ for each $v \in V$
    - ○ $u \to v$ for each $(u, v) \in E$

Bipartite Matching

- Observation
  - There is a 1-1 correspondence between matchings of size k in the original graph and flows with value k in the corresponding flow network.

- Proof: (matching $\Rightarrow$ integral flow)
  - Take a matching $M = \{(u_1, v_1), \dots, (u_k, v_k)\}$ of size k
  - Construct the corresponding unique flow f_M where…
    - Edges $s \to u_i$, $u_i \to v_i$, and $v_i \to t$ have flow 1, for all $i = 1, \dots, k$
    - The rest of the edges have flow 0
  - This flow has value k

Bipartite Matching

- Observation
  - There is a 1-1 correspondence between matchings of size k in the original graph and flows with value k in the corresponding flow network.

- Proof: (integral flow $\Rightarrow$ matching)
  - Take any flow f with value k
  - The corresponding unique matching $M_f =$ set of edges from U to V with a flow of 1
    - Since flow of k comes out of s, unit flow must go to k distinct vertices in U
    - From each such vertex in U, unit flow goes to a distinct vertex in V
    - Uses integrality theorem

Bipartite Matching

- Perfect matching = flow with value n
  - where $n = |U| = |V|$

- Recall naïve Ford-Fulkerson running time:
  - $O((m+n) \cdot C)$, where $C =$ sum of capacities of edges leaving s
  - Q: What's the runtime when used for bipartite matching?

- Some variants are faster…
  - Dinitz's algorithm runs in time $O(m\sqrt{n})$ when all edge capacities are 1

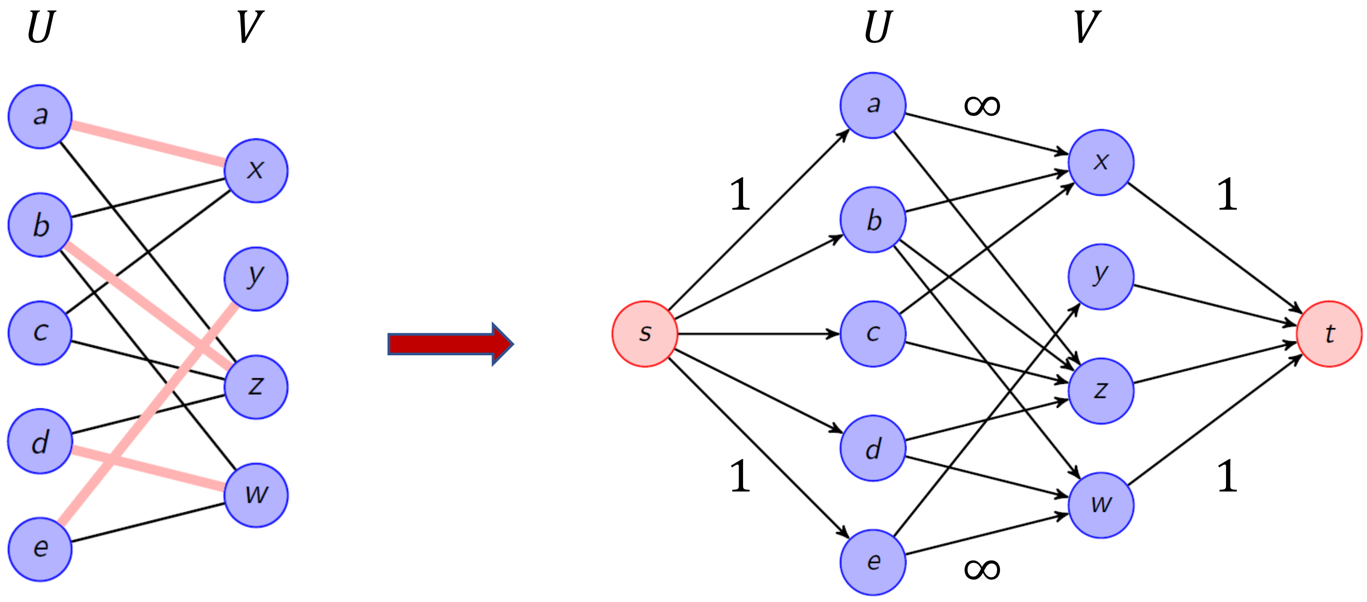
Hall's Marriage Theorem

- When does a bipartite graph have a perfect matching?
  - Well, when the corresponding flow network has value n
  - But can we interpret this condition in terms of edges of the original bipartite graph?
  - For $S \subseteq U$, let $N(S) \subseteq V$ be the set of all nodes in V adjacent to some node in S

- Observation:
  - If G has a perfect matching, $|N(S)| \geq |S|$ for each $S \subseteq U$
  - Because each node in S must be matched to a distinct node in $N(S)$

Hall's Marriage Theorem

- We'll consider a slightly different flow network, which is still equivalent to bipartite matching
  - All $U \to V$ edges now have ∞ capacity
  - $s \to U$ and $V \to t$ edges are still unit capacity



Hall's Marriage Theorem

- Hall's Theorem:
  - G has a perfect matching iff $|N(S)| \geq |S|$ for each $S \subseteq V$

- Proof (reverse direction, via network flow):
  - Suppose G doesn't have a perfect matching
  - Hence, max-flow = min-cut $< n$
  - Let (A, B) be the min-cut
    - Can't have any $U \rightarrow V$ (∞ capacity edges)
    - Has unit capacity edges $s \rightarrow U \cap B$ and $V \cap A \rightarrow t$

Hall's Marriage Theorem

- Hall's Theorem:
  - G has a perfect matching iff $|N(S)| \geq |S|$ for each $S \subseteq V$

- Proof (reverse direction, via network flow):
  - $cap(A, B) = |U \cap B| + |V \cap A| < n = |U|$
  - So $|V \cap A| < |U \cap A|$
  - But $N(U \cap A) \subseteq V \cap A$ because the cut doesn't include any ∞ edges
  - So $|N(U \cap A)| \leq |V \cap A| < |U \cap A|$. ■

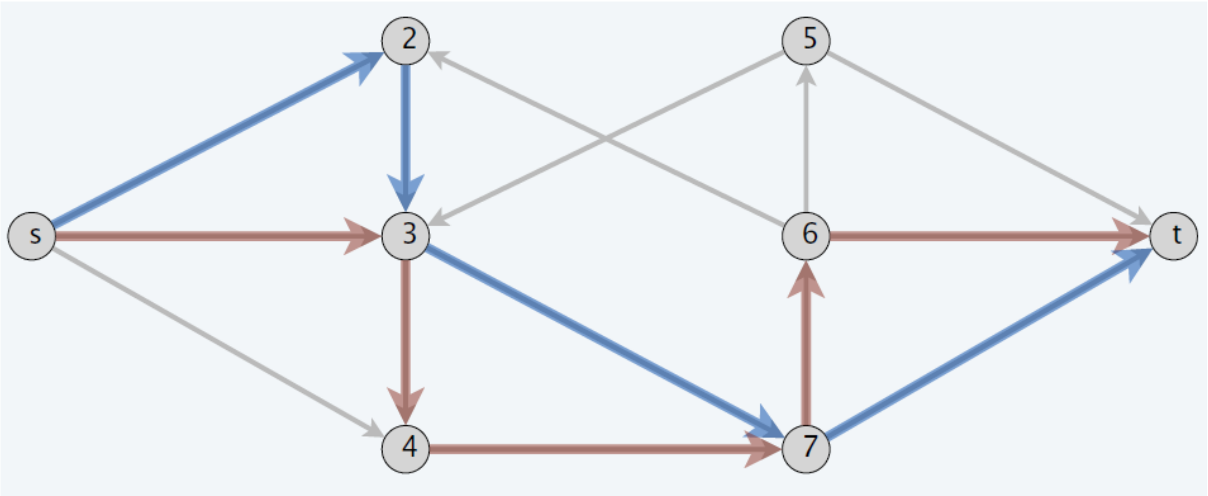
Some Notes

- Runtime for bipartite perfect matching
  - 1955: $O(mn) \rightarrow$ Ford-Fulkerson
  - 1973: $O(m\sqrt{n}) \rightarrow$ blocking flow (Hopcroft-Karp, Karzanov)
  - 2004: $O(n^{2.378}) \rightarrow$ fast matrix multiplication (Mucha–Sankowsi)
  - 2013: $\tilde{O}\left(m^{10/7}\right) \rightarrow$ electrical flow (Mądry)
  - Best running time is still an open question
- Nonbipartite graphs
  - Hall's theorem $\rightarrow$ Tutte's theorem
  - 1965: $O(n^4) \rightarrow$ Blossom algorithm (Edmonds)
  - 1980/1994: $O(m\sqrt{n}) \rightarrow$ Micali-Vazirani

Edge-Disjoint Paths

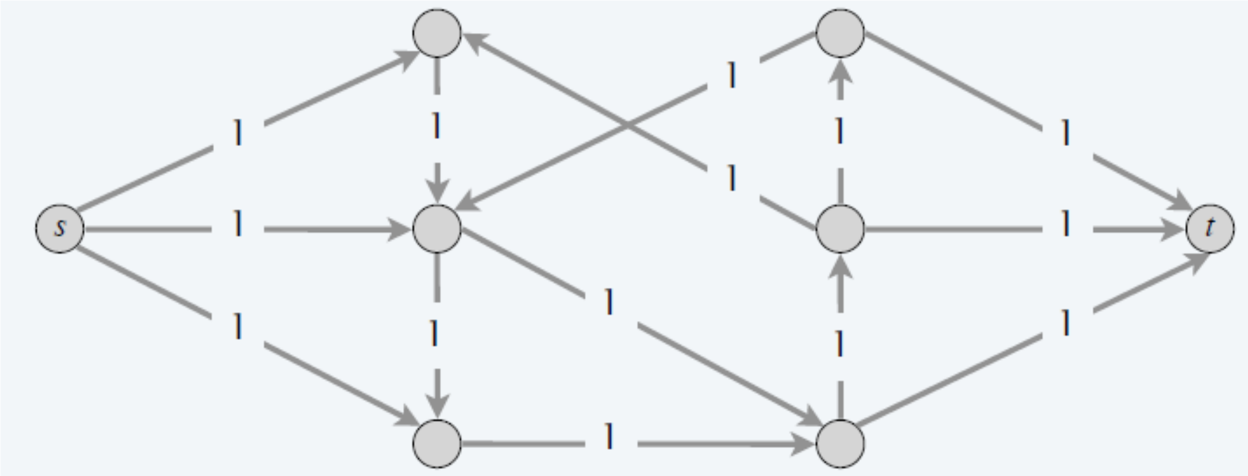
- Problem
  - Given a directed graph $G = (V, E)$, two nodes s and t, find the maximum number of edge-disjoint $s \to t$ paths

- Two $s \to t$ paths P and P' are edge-disjoint if they don't share an edge



Edge-Disjoint Paths

- Application:
  - Communication networks
- Max-flow formulation
  - Assign unit capacity on all edges



Edge-Disjoint Paths

- Theorem:
  - There is 1-1 correspondence between sets of k edge-disjoint $s \to t$ paths and integral flows of value k

- Proof (paths → flow)
  - Let $\{P_1, \dots, P_k\}$ be a set of k edge-disjoint $s \to t$ paths
  - Define flow f where $f(e) = 1$ whenever $e \in P_i$ for some i, and 0 otherwise
  - Since paths are edge-disjoint, flow conservation and capacity constraints are satisfied
  - Unique integral flow of value k

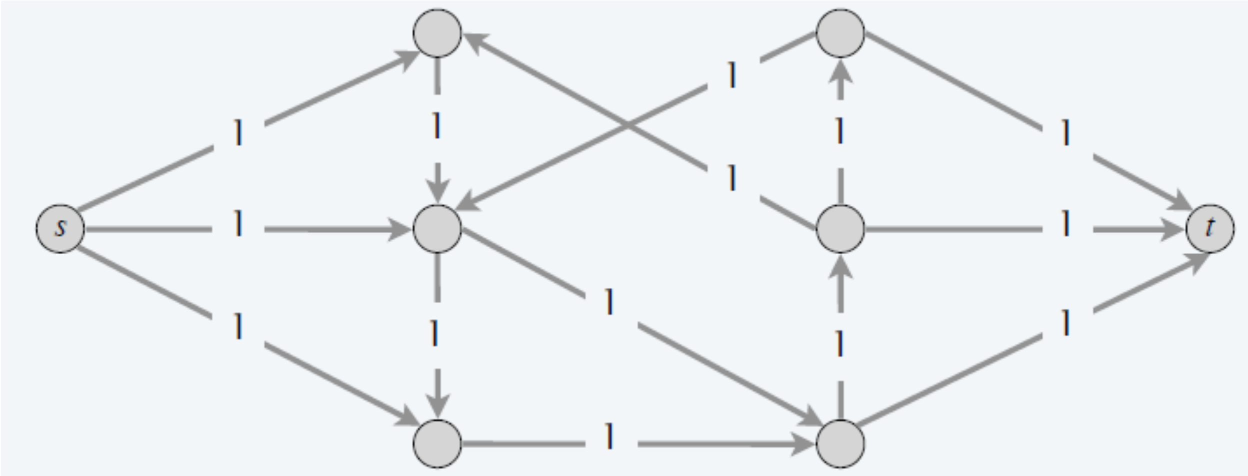
Edge-Disjoint Paths

- Theorem:
  - There is 1-1 correspondence between k edge-disjoint $s \to t$ paths and integral flows of value k

- Proof (flow $\to$ paths)
  - Let f be an integral flow of value k
  - k outgoing edges from s have unit flow
  - Pick one such edge (s, u_1)
    - By flow conservation, u_1 must have unit outgoing flow (which we haven't used up yet).
    - Pick such an edge and continue building a path until you hit t
  - Repeat this for the other $k - 1$ edges from s with unit flow ■

Edge-Disjoint Paths

- Maximum number of edge-disjoint $s \rightarrow t$ paths
  - Equals max flow in this network
  - By max-flow min-cut theorem, also equals minimum cut
  - Exercise: minimum cut = minimum number of edges we need to delete to disconnect s from t
    - Hint: Show each direction separately ($\leq$ and $\geq$)



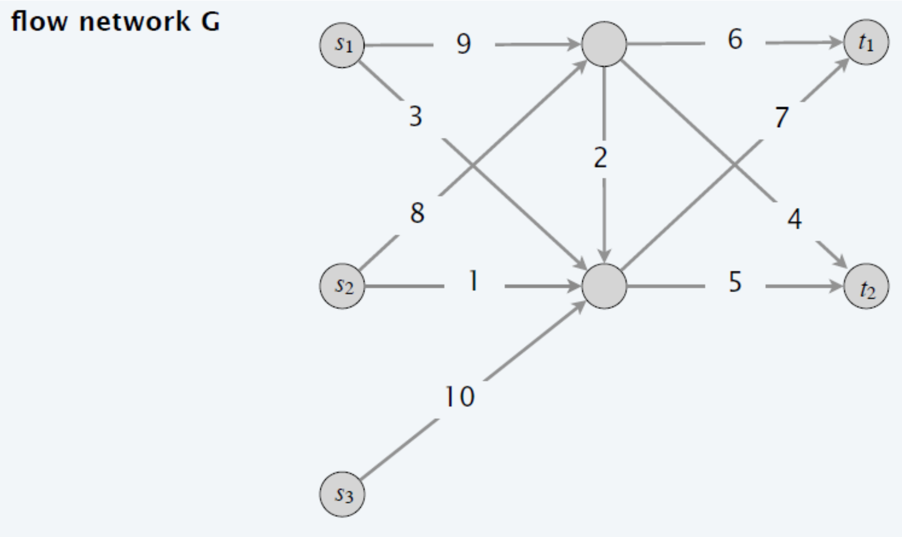
Edge-Disjoint Paths

- Exercise!
  - Show that to compute the maximum number of edge-disjoint s-t paths in an undirected graph, you can create a directed flow network by adding each undirected edge in both directions and setting all capacities to 1

- Menger's Theorem
  - In any directed/undirected graph, the maximum number of edge-disjoint (resp. vertex-disjoint) $s \to t$ paths equals the minimum number of edges (resp. vertices) whose removal disconnects s and t

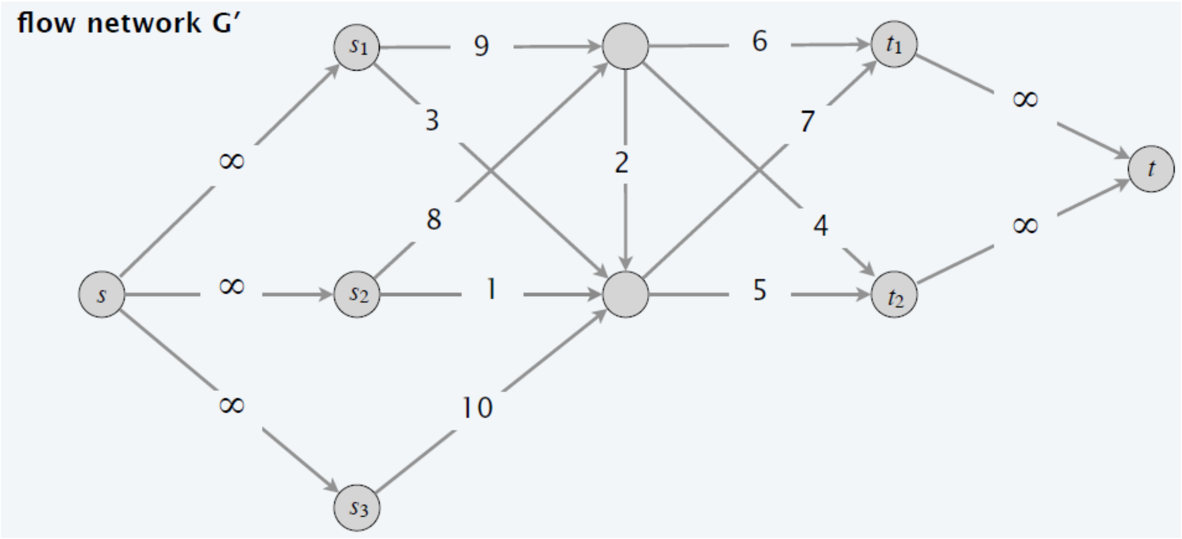
Multiple Sources/Sinks

- Problem
  - Given a directed graph $G = (V, E)$ with edge capacities $c: E \to \mathbb{N}$, sources $s_1, \dots, s_k$ and sinks $t_1, \dots, t_\ell$, find the maximum total flow from sources to sinks.



Multiple Sources/Sinks

- Network flow formulation
  - Add a new source s, edges from s to each s_i with ∞ capacity
  - Add a new sink t, edges from each t_j to t with ∞ capacity
  - Find max-flow from s to t
  - Claim: $1 - 1$ correspondence between flows in two networks



Circulation

- Input
  - Directed graph $G = (V, E)$
  - Edge capacities $c : E \to \mathbb{N}$
  - Node demands $d : V \to \mathbb{Z}$
- Output
  - Some circulation $f : E \to \mathbb{N}$ satisfying
    - For each $e \in E : 0 \leq f(e) \leq c(e)$
    - For each $v \in V : \sum_{e \text{ entering } v} f(v) - \sum_{e \text{ leaving } v} f(v) = d(v)$

- Note that you need $\sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$
- What are demands?

Circulation

- Demand at v = amount of flow you need to take out at node v

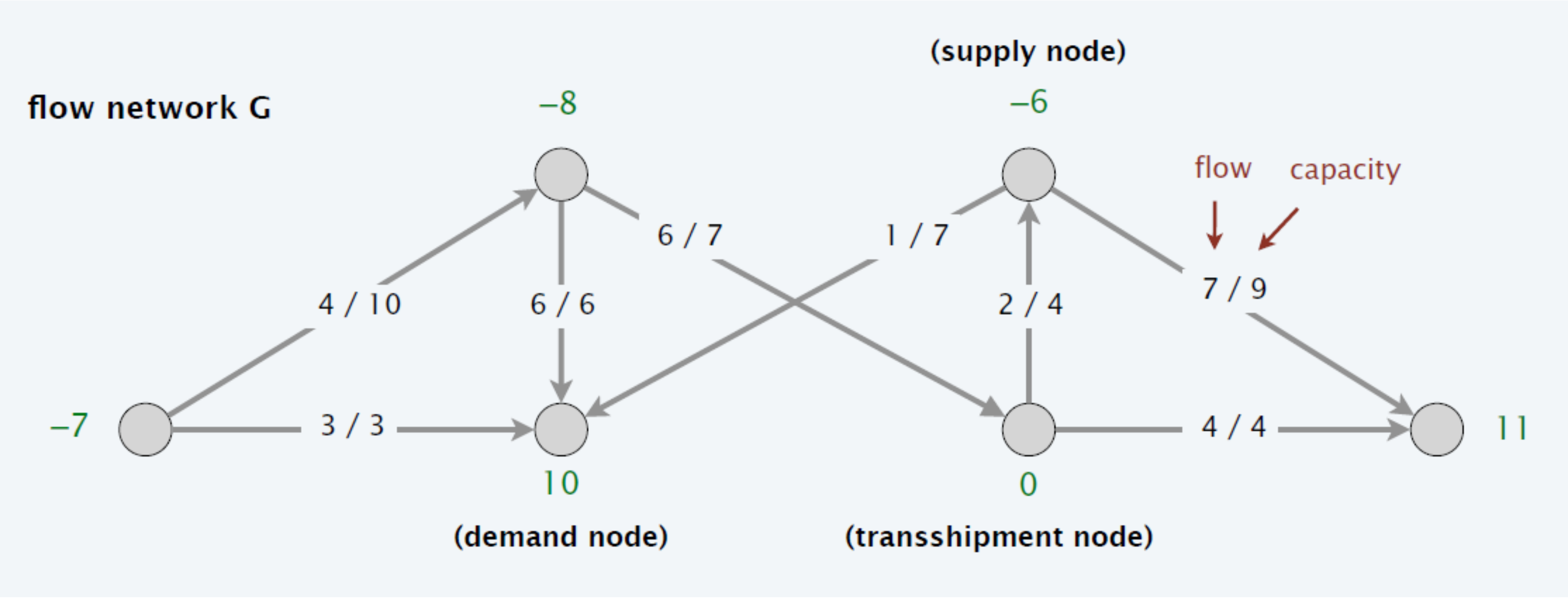
  - $d(v) > 0$: You need to take some flow out at v
    - So, there should be $d(v)$ *more* incoming flow than outgoing flow
    - "Demand node"

  - $d(v) < 0$: You need to put some flow in at v
    - So, there should be $|d(v)|$ *more* outgoing flow than incoming flow
    - "Supply node"

  - $d(v) = 0$: Node has flow conservation
    - Equal incoming and outgoing flows
    - "Transshipment node"

Circulation

- Example



Circulation

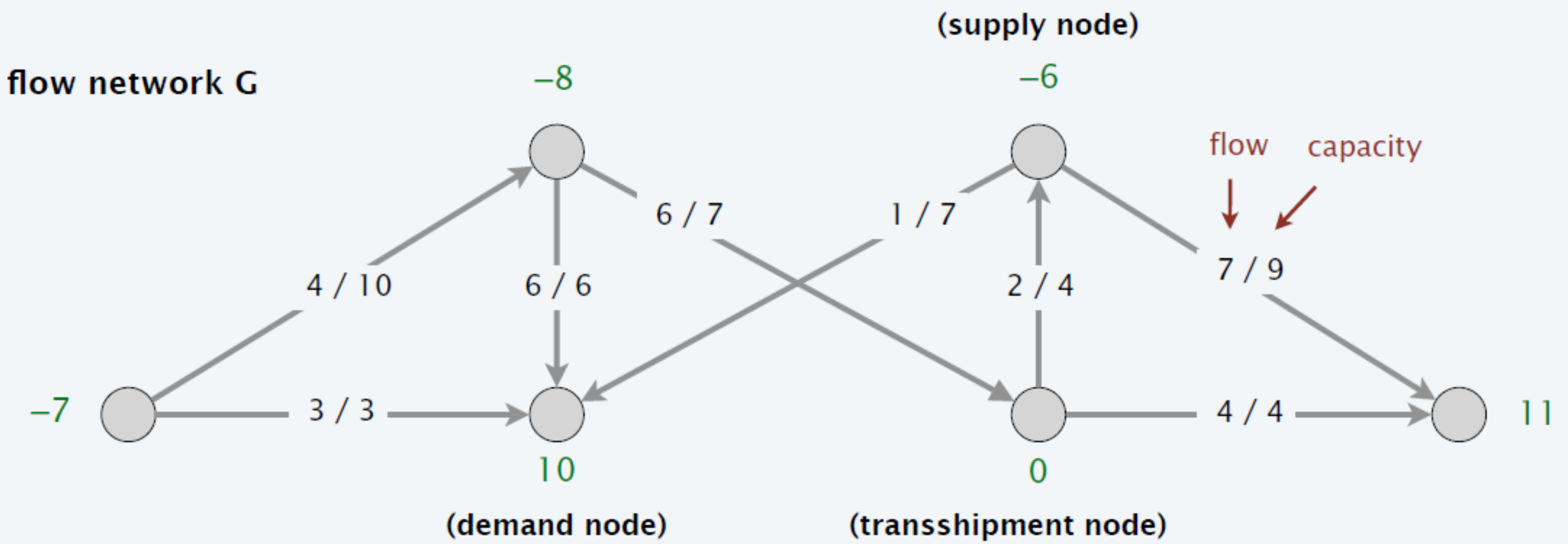
- Network-flow formulation G'
  - Add a new source s and a new sink t
  - For each "supply" node v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$
  - For each "demand" node v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$

- Claim:
  - G has a circulation iff G' has max flow of value

$$\sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$$

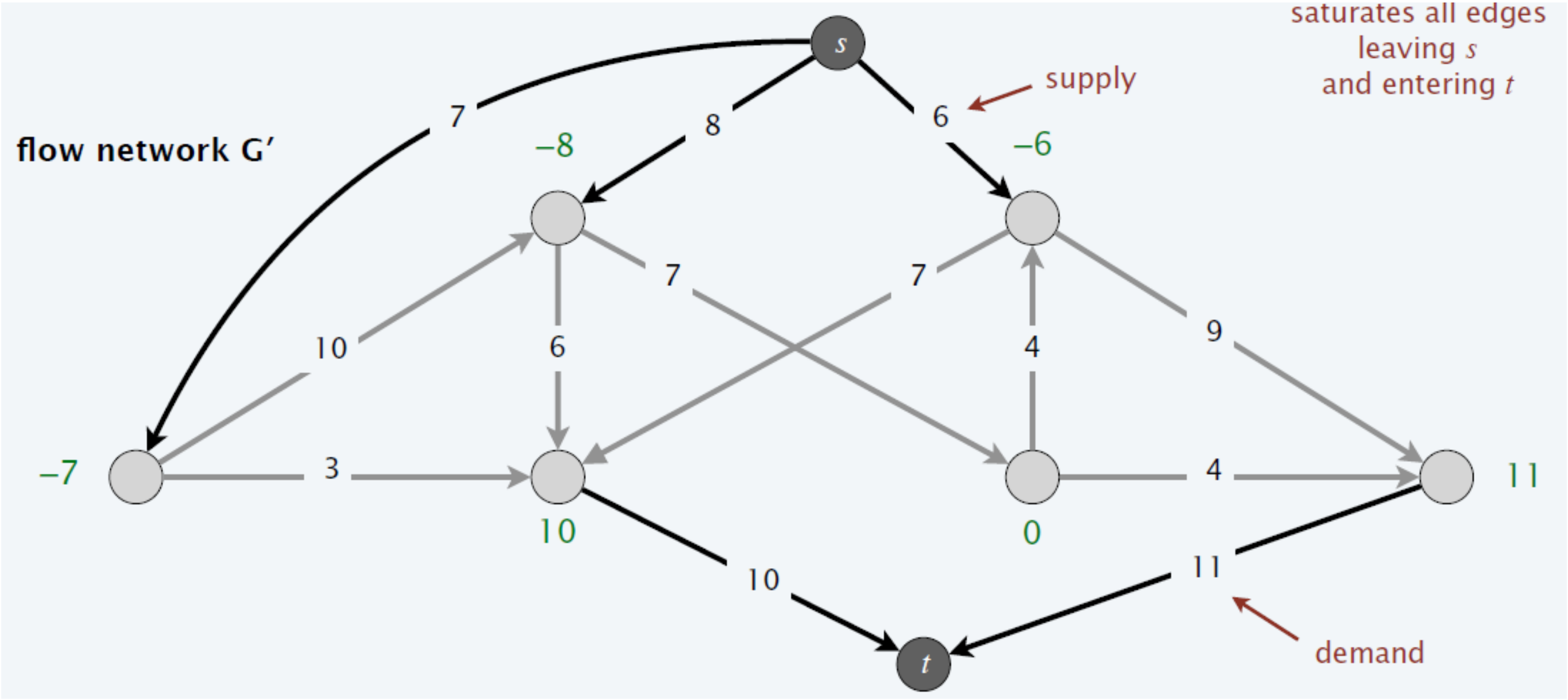
Circulation

- Example



Circulation

- Example



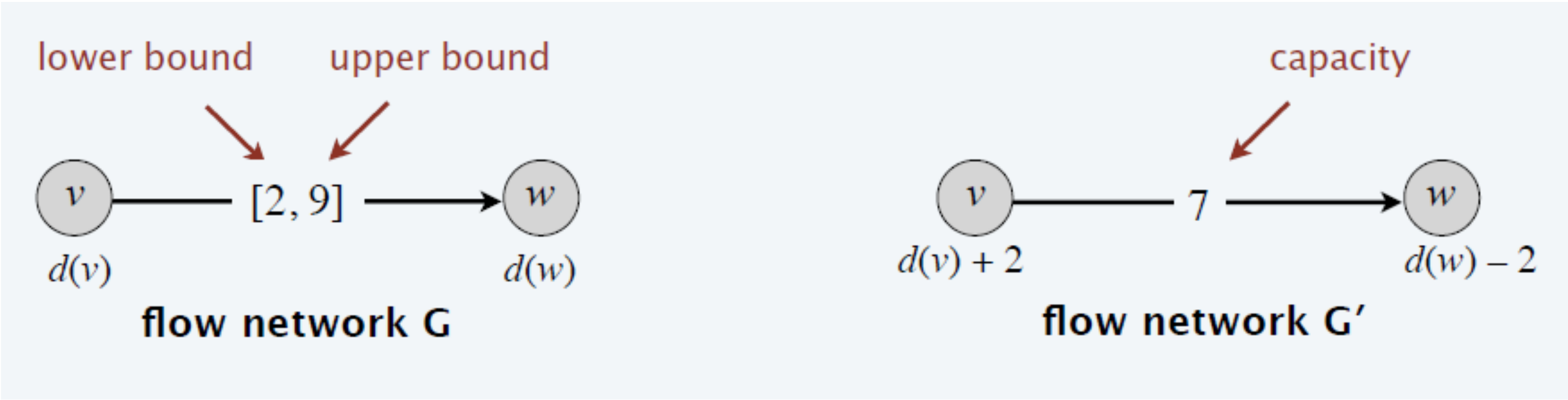
Circulation with Lower Bounds

- Input
  - Directed graph $G = (V, E)$
  - Edge capacities $c : E \to \mathbb{N}$ and lower bounds $\ell : E \to \mathbb{N}$
  - Node demands $d : V \to \mathbb{Z}$
- Output
  - Some circulation $f : E \to \mathbb{N}$ satisfying
    - For each $e \in E : \ell(e) \leq f(e) \leq c(e)$
    - For each $v \in V : \sum_{e \text{ entering } v} f(v) - \sum_{e \text{ leaving } v} f(v) = d(v)$

- Note that you still need $\sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$

Circulation with Lower Bounds

- Transform to circulation without lower bounds
  - Do the following operation to each edge



- Claim: Circulation in G iff circulation in G'
  - Proof sketch: $f(e)$ gives a valid circulation in G iff $f(e) - \ell(e)$ gives a valid circulation in G'

Survey Design

- Problem
  - We want to design a survey about m products
    - We have one question in mind for each product
    - Need to ask product j's question to between p_j and p_j' consumers
  - There are a total of n consumers
    - Consumer i owns a subset of products O_i
    - We can ask consumer i questions about only these products
    - We want to ask consumer i between c_i and c_i' questions
  - Is there a survey meeting all these requirements?

Survey Design

- Bipartite matching is a special case
  - $c_i = c_i' = p_j = p_j' = 1$ for all i and j

- Formulate as circulation with lower bounds
  - Create a network with special nodes s and t
  - Edge from s to each consumer i with flow $\in [c_i, c_i']$
  - Edge from each consumer i to each product $j \in O_i$ with flow $\in [0,1]$
  - Edge from each product j to t with flow $\in [p_j, p_j']$
  - Edge from t to s with flow in $[0, \infty]$
  - All demands and supplies are 0

Survey Design

- Max-flow formulation:
  - Feasible survey iff feasible circulation in this network

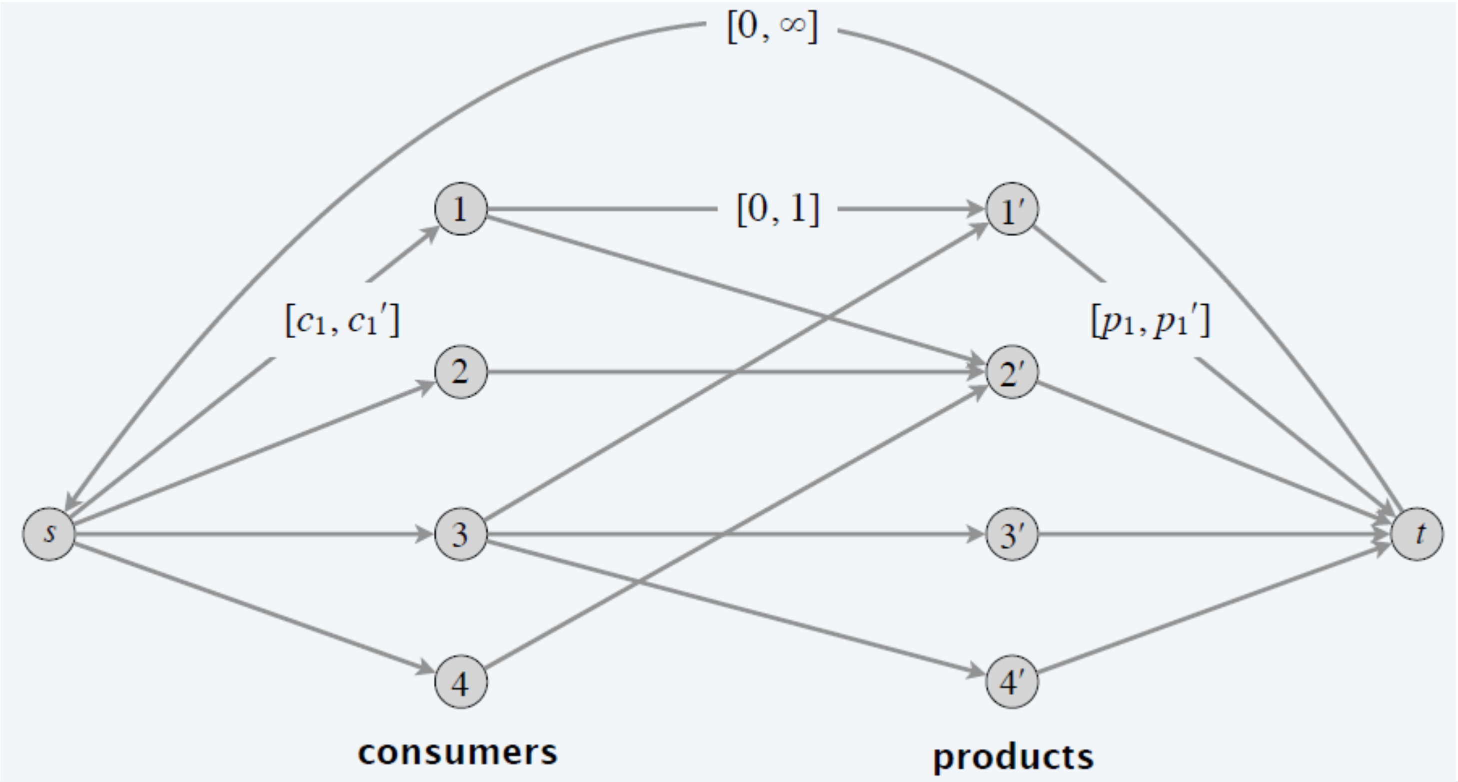


Image Segmentation

- Foreground/background segmentation
  - Given an image, separate "foreground" from "background"
- Here's the power of PowerPoint (or the lack thereof)

Remove background

Image Segmentation

- Foreground/background segmentation
  - Given an image, separate "foreground" from "background"
- Here's what remove.bg gets using AI

Remove background

Image Segmentation

- Informal problem
  - Given an image (2D array of pixels), and likelihood estimates of different pixels being foreground/background, label each pixel as foreground or background
  - Want to prevent having too many neighboring pixels where one is labeled foreground but the other is labeled background

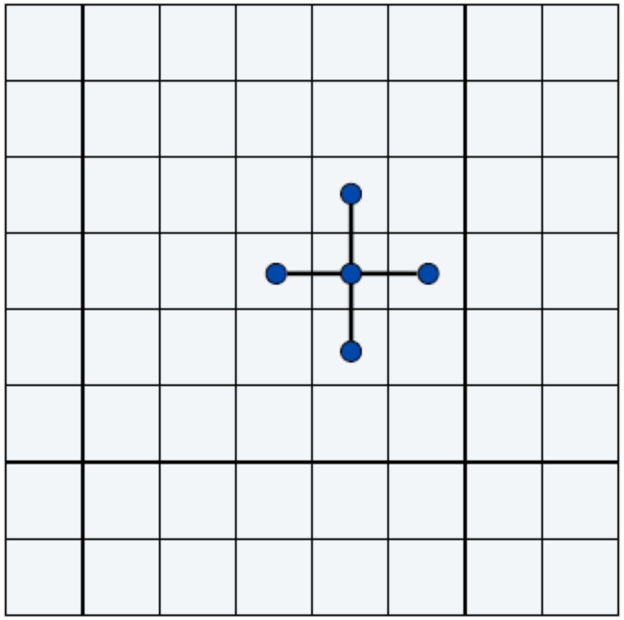


Image Segmentation

- Input
  - An image (2D array of pixels)
  - a_i = likelihood of pixel i being in foreground
  - b_i = likelihood of pixel i being in background
  - $p_{i,j}$ = penalty for "separating" pixels i and j (i.e. labeling one of them as foreground and the other as background)
- Output
  - Label each pixel as "foreground" or "background"
  - Minimize "total penalty"
    - Want it to be high if a_i is high but i is labeled background, b_i is high but i is labeled foreground, or $p_{i,j}$ is high but i and j are separated

Image Segmentation

- Recall
  - a_i = likelihood of pixels i being in foreground
  - b_i = likelihood of pixels i being in background
  - $p_{i,j}$ = penalty for separating pixels i and j
  - Let $E =$ pairs of neighboring pixels
- Output
  - Minimize total penalty
    - $A =$ set of pixels labeled foreground
    - $B =$ set of pixels labeled background
    - Penalty =

$$\sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{i,j}$$

Image Segmentation

- Formulate as a min-cut problem
  - Want to divide the set of pixels V into (A, B) to minimize

$$\sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{i,j}$$

  - Nodes:
    - source s, target t, and v_i for each pixel i
  - Edges:
    - (s, v_i) with capacity a_i for all i
    - (v_i, t) with capacity b_i for all i
    - (v_i, v_j) and (v_j, v_i) with capacity $p_{i,j}$ each for all neighboring (i, j)

Image Segmentation

- Formulate as min-cut problem
  - Here's what the network looks like

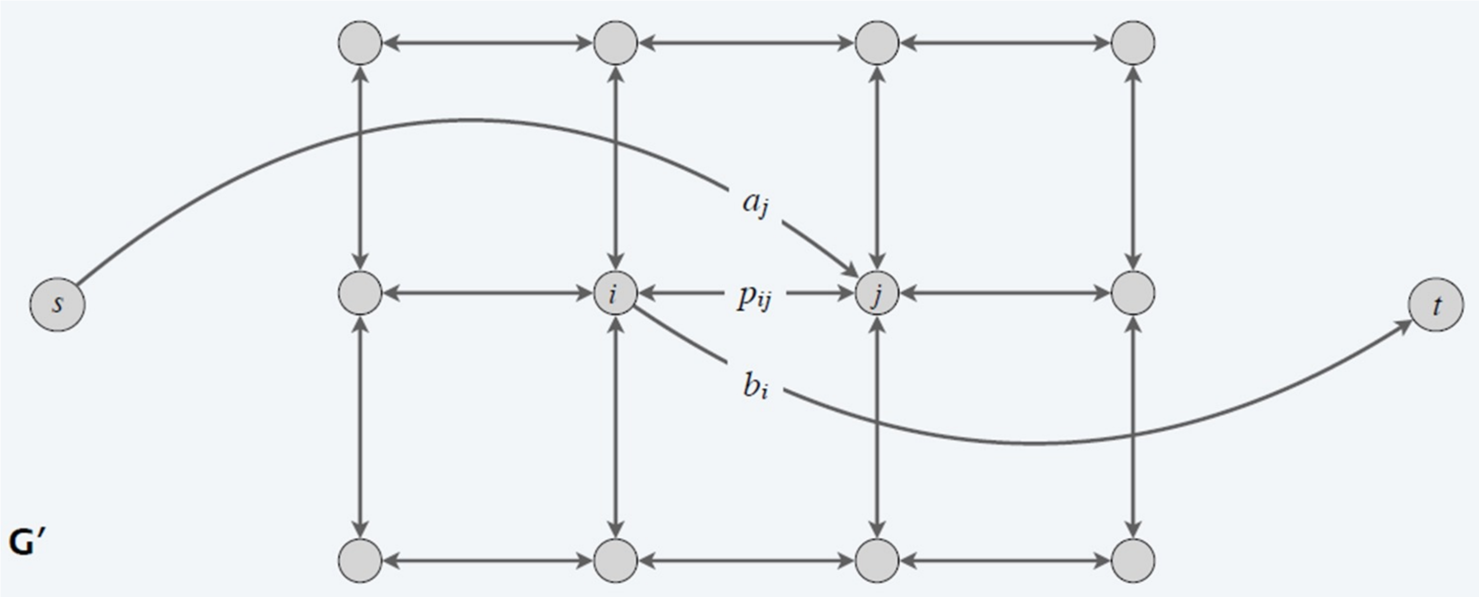


Image Segmentation

If i and j are labeled differently, it will add $p_{i,j}$ exactly once

- Consider the min-cut (A, B)

$$cap(A,B) = \sum_{i \in A} b_i + \sum_{j \in B} a_j + \sum_{\substack{(i,j) \in E \\ i \in A, j \in B}} p_{i,j}$$

- Exactly what we want to minimize!

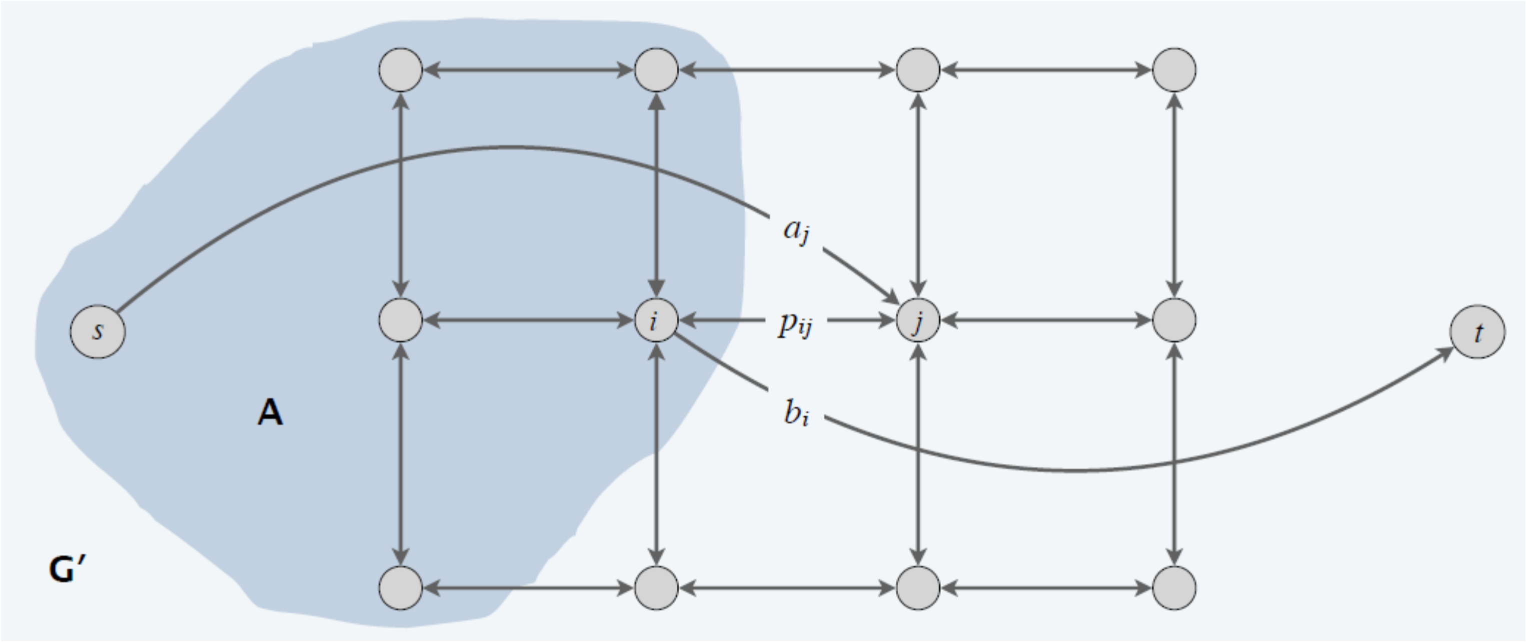


Image Segmentation

- GrabCut [Rother-Kolmogorov-Blake 2004]

“GrabCut” — Interactive Foreground Extraction using Iterated Graph Cuts

Carsten Rother* Vladimir Kolmogorov† Andrew Blake‡
Microsoft Research Cambridge, UK

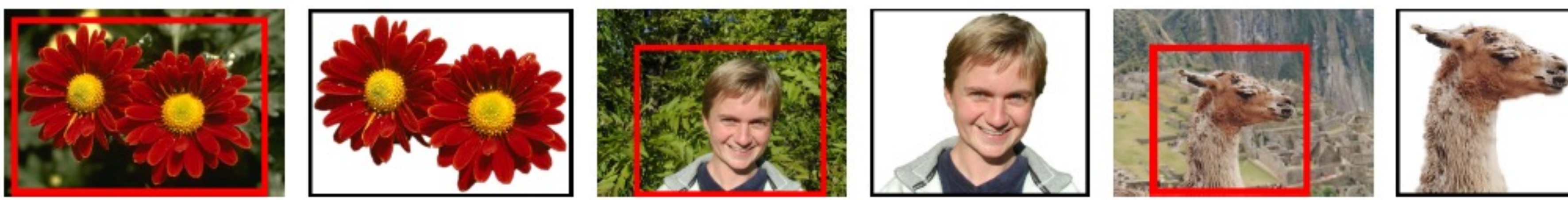


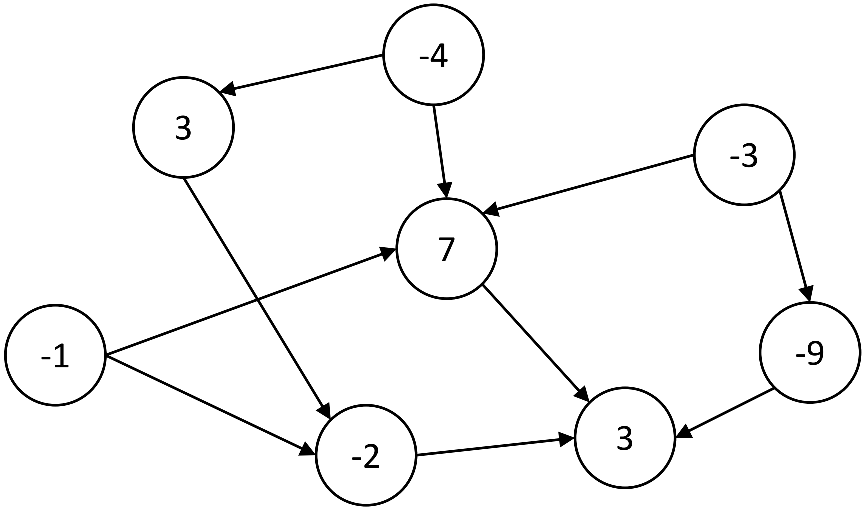
Figure 1: **Three examples of GrabCut .** The user drags a rectangle loosely around an object. The object is then extracted automatically.

Profit Maximization (Yeaa…!)

- Problem
  - There are n tasks
  - Performing task i generates a profit of p_i
    - We allow $p_i < 0$ (i.e., performing task i may be costly)
  - There is a set E of precedence relations
    - $(i, j) \in E$ indicates that if we perform i, we must also perform j
- Goal
  - Find a subset of tasks S which, subject to the precedence constraints, maximizes $profit(S) = \sum_{i \in S} p_i$

Profit Maximization

- We can represent the input as a graph
  - Nodes = tasks, node weights = profits,
  - Edges = precedence constraints
  - Goal: find a subset of nodes S with highest total weight s.t. if $i \in S$ and $(i, j) \in E$, then $j \in S$ as well

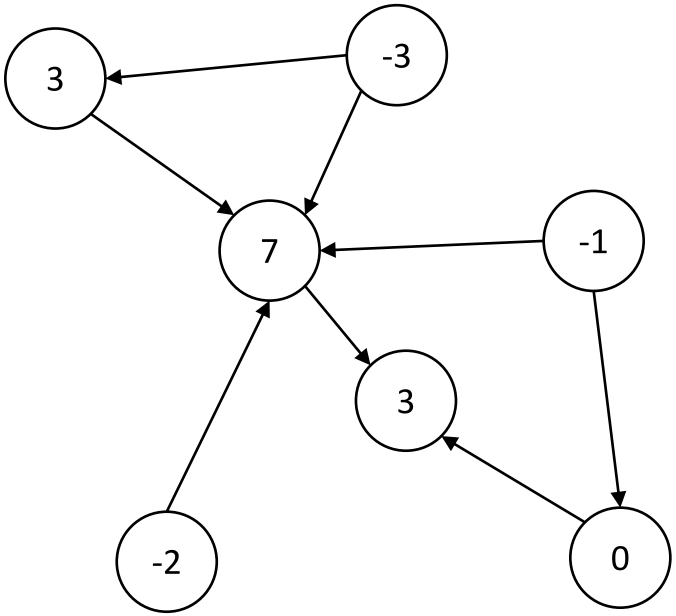


Profit Maximization

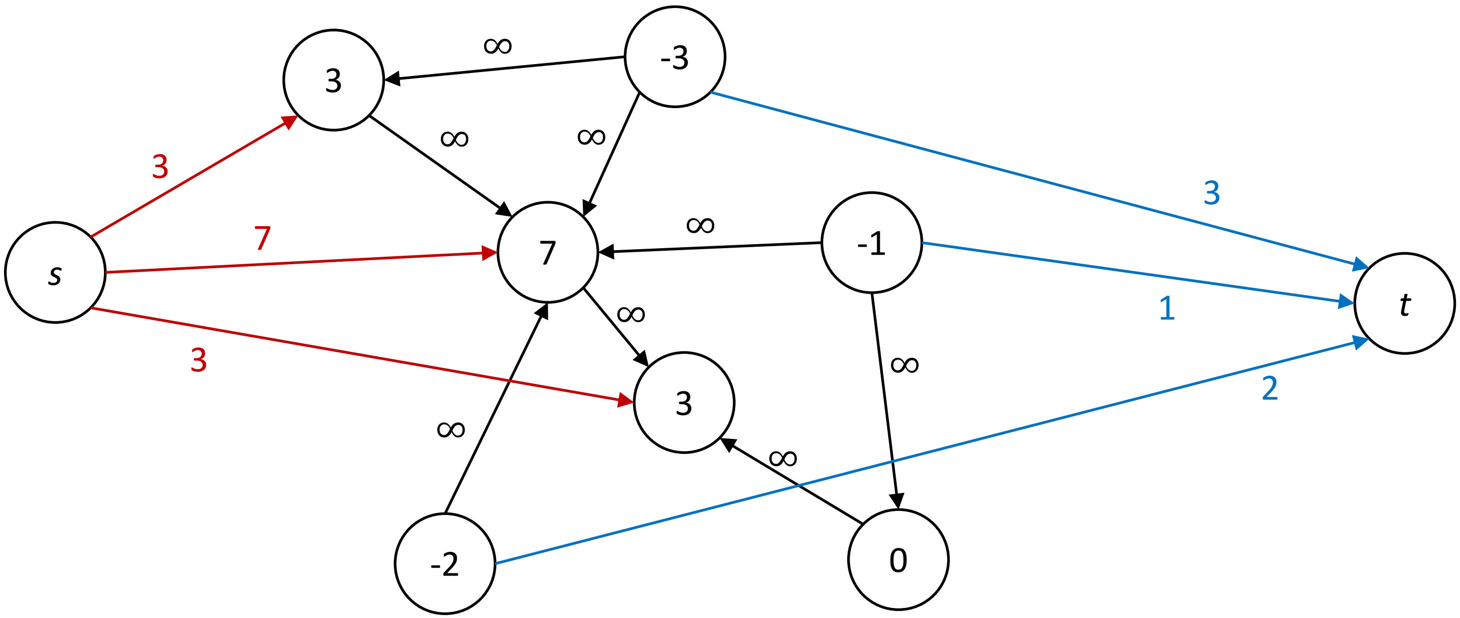
- Want to formulate as a min-cut
  - Add source s and target t
  - min-cut $(A, B) \Rightarrow$ want desired solution to be $S = A \setminus \{s\}$
  - Goals:
    - $cap(A, B)$ should nicely relate to $profit(S)$
    - Precedence constraints *must be* respected
      - "Hard" constraints are usually enforced using infinite capacity edges

- Construction:
  - Add each $(i, j) \in E$ with *infinite* capacity
  - For each i:
    - If $p_i > 0$, add (s, i) with capacity p_i
    - If $p_i < 0$, add (i, t) with capacity $-p_i$

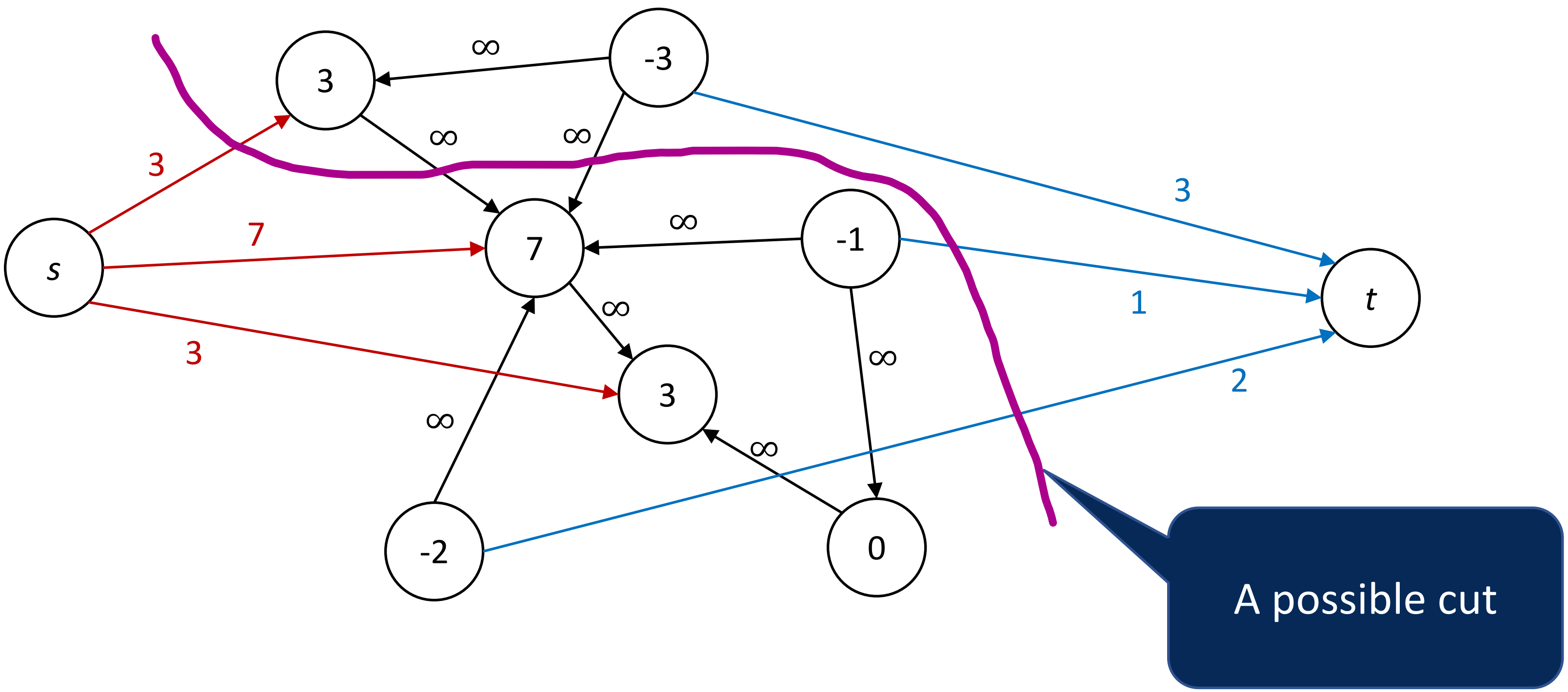
Profit Maximization



Profit Maximization



Profit Maximization



QUESTION: What is the capacity of this cut?

Profit Maximization

Exercise: Show that…

1. A finite capacity cut exists.
2. If $cap(A, B)$ is finite, then $A \backslash \{s\}$ is a valid solution;
3. Minimizing $cap(A, B)$ maximizes $profit(A \backslash \{s\})$
   - Show that $cap(A, B) = \text{constant} - profit(A \backslash \{s\})$, where the constant is independent of the choice of (A, B)