Dynamic Programming and Network Flows

Admin

- Problem Set 4 back
- Problem Set 5 back tomorrow
  - (thanks to our wonderful TAs for helping me and having a very quick turnaround)
- I will post a handout on tips for Dynamic Programming consolidating some of what we've seen

Admin: TA items

- TA evaluation form! https://forms.gle/sbqCGVLAFnhUQ4i39
  - Please fill out by next Friday
- Please apply to be a TA next semester!
  - https://csci.williams.edu/tatutor-application/
  - Don’t need to any kind of “algorithms person.”
    - Good to have different perspectives!
    - Class will be a little different in any case
  - Great way to learn algorithms better!

Midterm

- In-person during class two weeks from today
    - Required to take it at that time
- Very strong focus on topics since last midterm:
    - Divide and conquer/recurrences
    - Dynamic programming
        - Remember: I’ll give you the recipe
    - Network flows
- Closed book, but you can bring a 1-page (2-sided) cheat sheet
    - I don’t think it will be *too* helpful
- Practice exam posted soon

Planning for Final

- Sunday, May 25th at 1:30pm
- I will hold an extra final during reading period May 17-20
  - Only one! If you miss this one you need to take it on the 25th
- Please let me know as soon as possible if you want to take the exam early
- Especially: please let me know if you have any conflicts in May 17-20.

Partitioning Work

- Suppose we have to scan through a shelf of books, and each book has a different size
- We want to divide the shelf into k region of books, and each region is assigned one of the workers
- Order of books fixed by cataloging system: cannot reorder/ rearrange the books
- **Goal**: divide the work is a fair way among the workers

Subproblem

- **Subproblem**

$M(i, j)$ be the optimal cost of partitioning elements $s_1, s_2, \ldots, s_i$ using j partitions, where $1 \leq i \leq n$, $1 \leq j \leq k$

- **Final answer**

$M(n, k)$

Towards a Recurrence

- Want a recurrence for $M(i, j)$
- Notice that the jth partition starts after we place the $(j-1)$st "divider"
- Where can we place the $j-1st$ divider?

$i' + 1, \ldots, i$ books in jth partition

$1, \ldots, i'$ books using $j-1$ partitions

Towards a Recurrence

- Where can we place the $j-1st$ divider?
    - Between books i' and $i'+1$ for some $i' < i$

$i'+1, \ldots, i$ books in jth partition

$1, \ldots, i'$ books using $j-1$ partitions

Towards a Recurrence

- Finally: for to choose the partition point i' for starting the jth partition
  - Let us consider all possibilities $1 \leq i' < i$
  - Take min cost option among them

$i' + 1, \ldots, i$ books in jth partition

$1, \ldots, i'$ books using $j - 1$ partitions

Final Recurrence

- For $2 \leq i \leq n$ and $2 \leq j \leq k$, we have:

$$M(i, j) = \min_{1 \leq i' < i} \text{cost of starting } j\text{th parition at book } (i' + 1)$$

Towards a Recurrence

- Cost of this way of partitioning?
  - (Remember cost is max sum across all partitions)

$i' + 1, \ldots, i$ books in jth partition

$1, \ldots, i'$ books using $j - 1$ partitions

Towards a Recurrence

- Cost of jth partition itself: $\sum_{t=i'+1}^{i} s_i$
- Cost of remaining partitions? $M[i', j-1]$

$i' + 1, \ldots, i$ books in jth partition

$1, \ldots, i'$ books using $j - 1$ partitions

Final Recurrence

- For $2 \leq i \leq n$ and $2 \leq j \leq k$, we have:

$$M(i, j) = \min_{1 \leq i' < i} \max\{M(i', j-1), \sum_{\ell = i'+1}^{i} s_t\}$$

- **Memoization structure**: We store $M[i, j]$ values in a 2-D array or table using space $O(nk)$
- **Evaluation order**: In what order should we fill in the table?

Final Pieces

- Evaluation order.
  - To fill out $M[i, j]$, I need the previous column filled in for rows less than i, that is, $M[i', j-1]$ for all $1 \leq i' < i$
  - Can compute using column major order: column by column
- Running time?
  - Size of table (space): $O(k \cdot n)$
  - How long to compute a single cell?
    - Depends on n other cells
    - $O(n)$ time to fill in one cell

Running Time

- Running time
    - $O(n^2 \cdot k)$
- Is this a polynomial running time?
    - Not as stated, not polynomial in the number of bits required to write k
    - But lets think if we can upper bound k using n
- How big can k get?
    - At most n non-empty partitions of n elements
    - $O(n^3)$ algorithm in the worst case

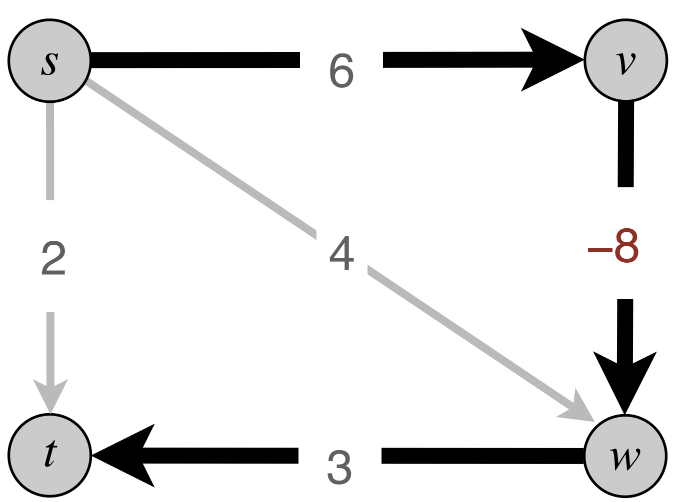
Last Topic in Dynamic Programming:
Shortest Paths Revisited

Shortest Path Problem

- **Single-Source Shortest Path Problem**.
  Given a connected directed graph $G = (V, E)$ with edge weights w_e on each $e \in E$ and a a source node s, find the shortest path from s to to all nodes in G.
- **Negative weights**. The edge-weights $w_e$in G can be negative. (When we studied Dijkstra's, we assumed non-negative weights.)
- Let P be a path from s to t, denoted $s \rightsquigarrow t$.
  - The **length** of P is the number of edges in P
  - The cost or weight of P is $w(P) = \sum_{e \in P} w_e$
- Goal: **cost** of the shortest path from s to all nodes

Negative Weights & Dijkstra's

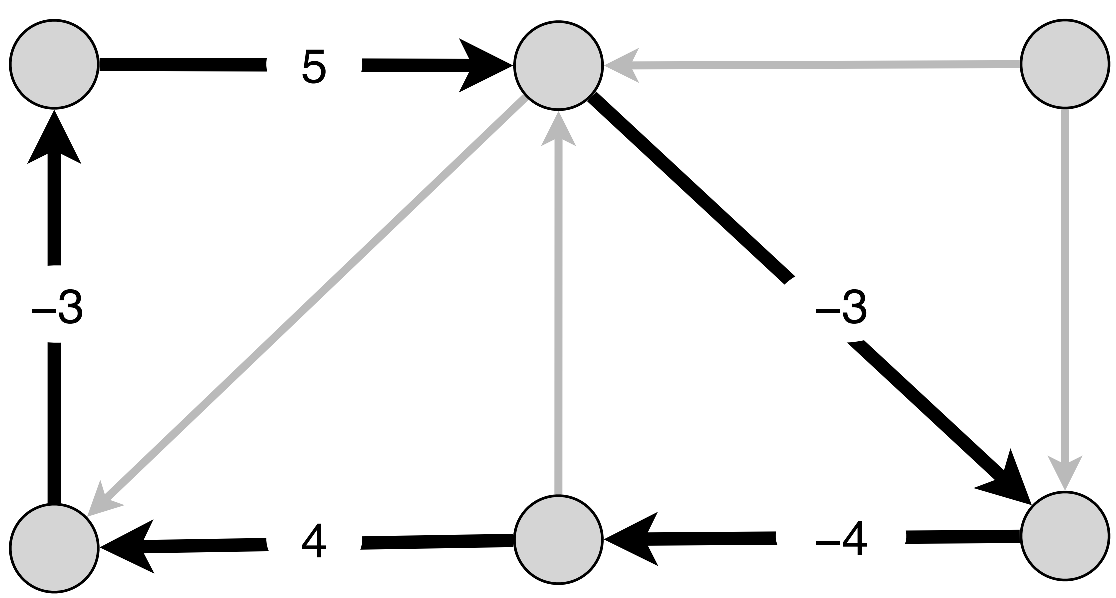
- **Dijkstra's Algorithm**. Does the greedy approach work for graphs with negative edge weights?
  - Dijkstra's will explore s's neighbor and add t, with $d[t] = w_{sv} = 2$ to the shortest path tree
  - Dijkstra assumes that there cannot be a "longer path" that has lower cost (relies on edge weights being non-negative)



Dijkstra's will find $s \to t$ as shortest path with cost 2
But the shortest path is $s \to v \to w \to t$ with cost 1

Negative Cycles

- **Definition**. A negative cycle is a directed cycle C such that the sum of all the edge weights in C is less than zero
- **Question**. How do negative cycles affect shortest path?



a negative cycle W : $\ell(W) = \sum_{e \in W} \ell_e < 0$

Negative Cycles & Shortest Paths

- **Claim.** If a path from s to some node v contains a negative cycle, then there does not exist a shortest path from s to v.

- **Proof**.
  - Suppose there exists a shortest $s \rightsquigarrow v$ path with cost d that traverses the negative cycle t times for $t \geq 0$.
  - Can construct a shorter path by traversing the cycle $t + 1$ times

  $\Rightarrow\!\!\Leftarrow$ ∎

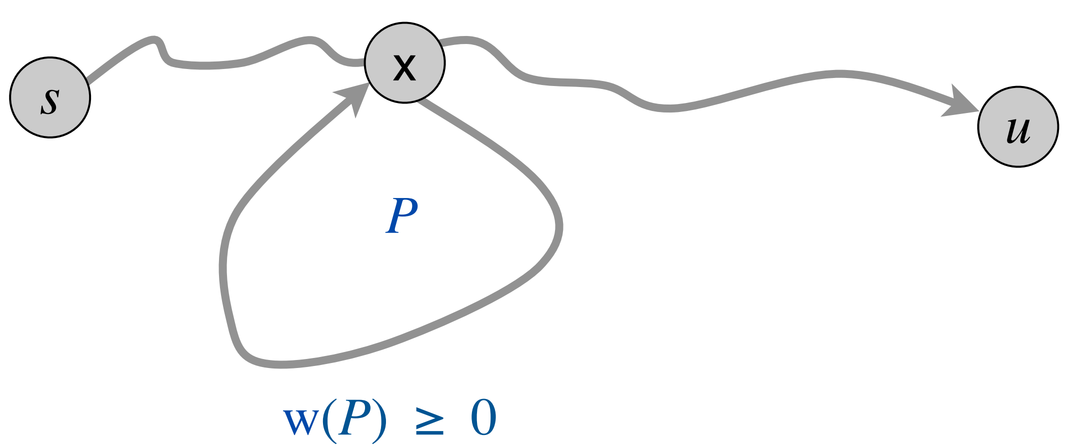
- **Assumption.** G has no negative cycle.
- Later in the lecture: how can we detect whether the input graph G contains a negative cycle?

Dynamic Programming Approach

- First step to a dynamic program? Recursive formulation
  - What is the subproblem? What is the recurrence?
  - Dijkstra's algorithm: for each v the subproblem is the shortest path from s to v
  - Why doesn't this work?
  - There may be a **shorter** path out of the cut (but it must have **more edges)**
  - **Idea:** subproblem (v, k) is the shortest path from s to v consisting of at most k edges
- How big can k get?

No. of Edges in Shortest Path

- **Claim.** If G has no negative cycles, then exists a shortest path from s to any node u that uses at most $n - 1$ edges.
- **Proof**. Suppose there exists a shortest path from s to u made up of n or more edges
- A path of length at least n must visit at least $n + 1$ nodes
- There exists a node x that is visited more than once (**pigeonhole principle**). Let P denote the portion of the path between the successive visits.
- Can remove P without increasing cost of path. ∎

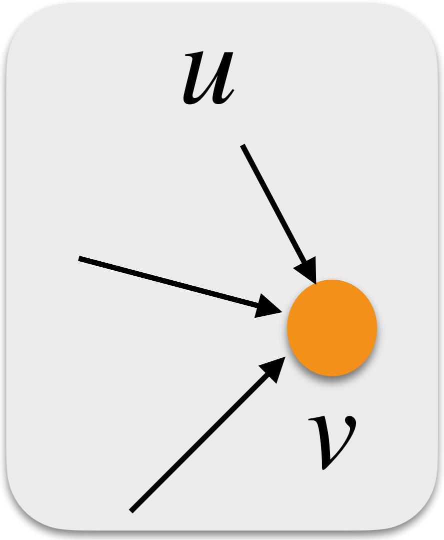


Shortest Path Subproblem

- **Subproblem**. $D[v, i]$: (optimal) cost of shortest path from s to v using $\leq i$ edges, or ∞ if no path with $\leq i$ edges
- **Base cases**.
  - $D[s, i] = 0$ for any i
  - $D[v, 0] = \infty$ for any $v \neq s$
- **Final answer** for shortest path cost to node v
  - $D[v, n - 1]$

Recurrence

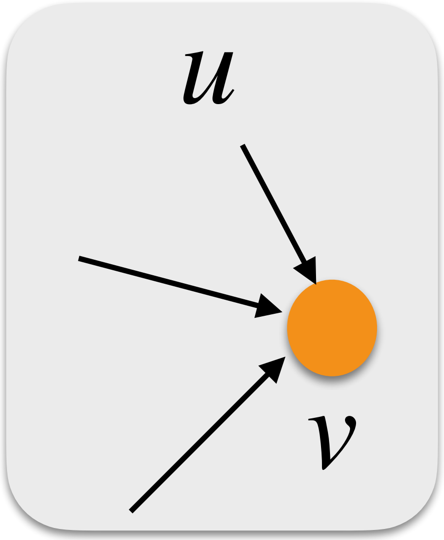
- Suppose we have found shortest paths to all nodes of length at most $i - 1$
- We are now considering shortest paths of length i
- Cases to consider for the **recurrence** of $D[v, i]$
  - **Case 1**. Shortest path to v was already found (is same as $D[v, i - 1]$)
  - **Case 2**. Shortest path to v is "longer" than paths found so far:
    - Look at all nodes u that have incoming edges to v
    - Take minimum over their distances and add w_{uv}



Bellman-Ford-Moore Algorithm

- **Recurrence.** For all nodes $v \neq s$, and for all $1 \leq i \leq n-1$,

$$D[v,i] = \min\{D[v,i-1], \min_{(u,v)\in E} \{D[u,i-1] + w_{uv}\}\}$$



- Called the **Bellman-Ford-Moore** algorithm

Bellman-Ford-Moore Algorithm

- **Subproblem**. $D[v, i]$: (optimal) cost of shortest path from s to v using $\leq i$ edges

- **Recurrence.**
  $$D[v, i] = \min\{D[v, i-1], \min_{(u,v)\in E} \{D[u, i-1] + w_{uv}\}\}$$

- **Memoization structure**. Two-dimensional array

- **Evaluation order**.
  - $i : 1 \rightarrow n-1$ (column major order)
  - Starting from s, the row of vertices can be in any order

Running Time

- **Recurrence**.
  $D[v, i] = \min\{D[v, i-1], \min_{(u,v)\in E} \{D[u, i-1] + w_{uv}\}\}$

- **Naive analysis**. $O(n^3)$ time
  - Each entry takes $O(n)$ to compute, there are $O(n^2)$ entries

- **Improved analysis**. For a given i, v, $d[v, i]$ looks at each incoming edge of v
  - Takes indegree(v) accesses to the table
  - For a given i, filling $d[-, i]$ takes $\sum_{v\in V}$ indegree(v) accesses
  - At most $O(n + m) = O(m)$ accesses (remember that for connected graphs we have $m \geq n - 1$)

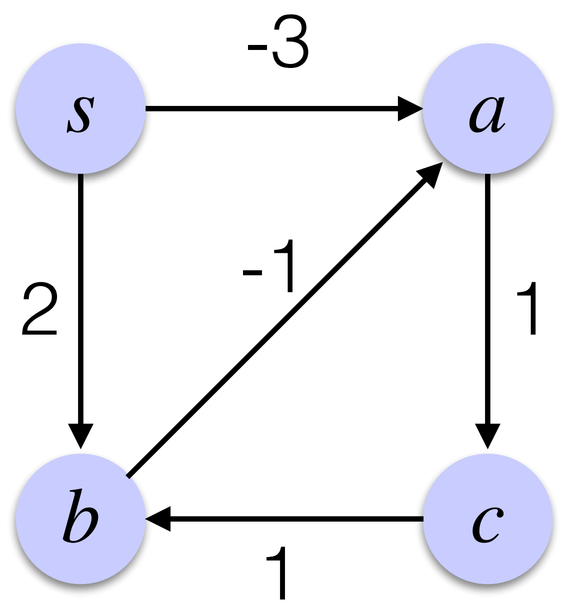
- Overall running time is $O(nm)$

- **Shortest-Path Summary.** Assuming there are no negative cycles in G, we can compute the shortest path from s to all nodes in G in $O(nm)$ time using the Bellman-Ford-Moore algorithm

Dynamic Programming
Shortest Path:
Bellman-Ford-Moore Example

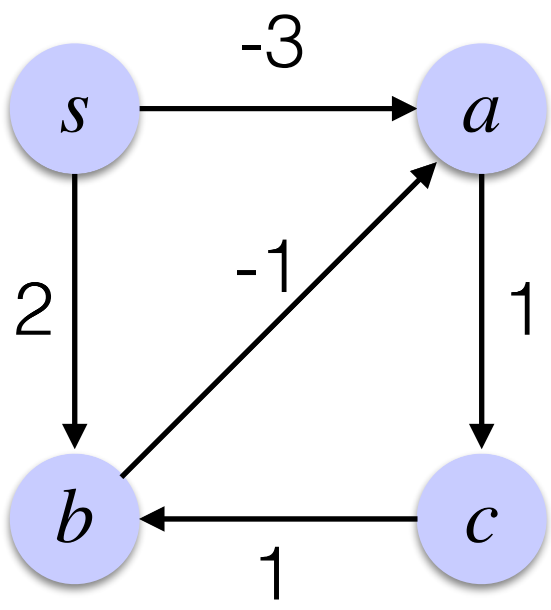
- $D[s, i] = 0$ for any i
- $D[v, 0] = \infty$ for any $v \neq s$

	0	1	2	3
s	0	0	0	0
a	inf			
b	inf			
c	inf			



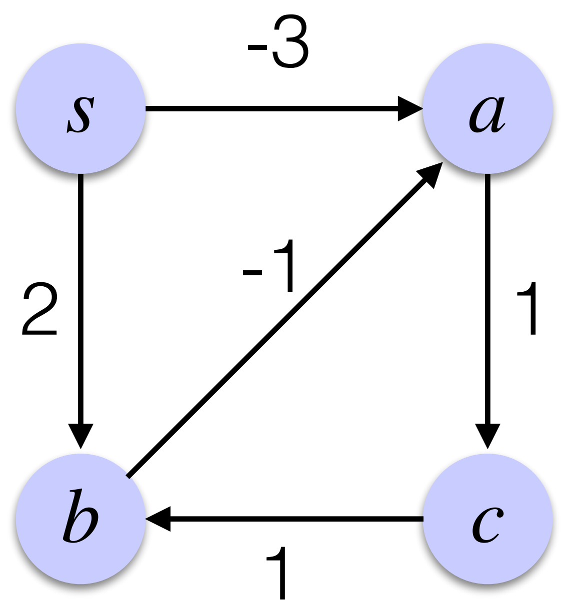
- $D[v,1] = \min\{D[v,0], \min_{u,v \in E} \{D[u,0] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf			
b	inf			
c	inf			



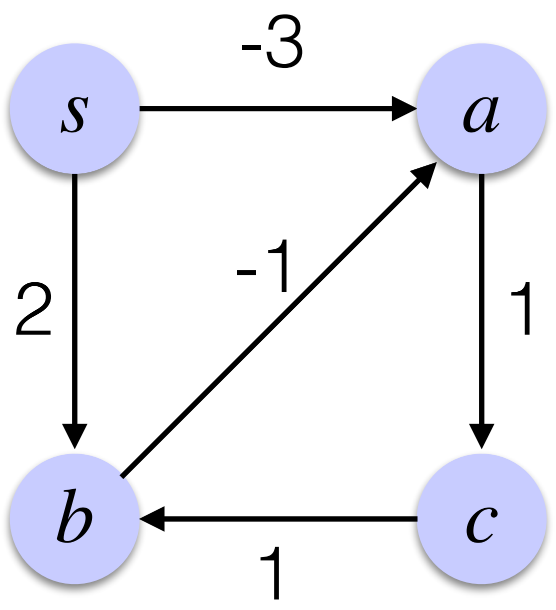
- $D[v,1] = \min\{D[v,0], \min_{u,v \in E} \{D[u,0] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3		
b	inf			
c	inf			



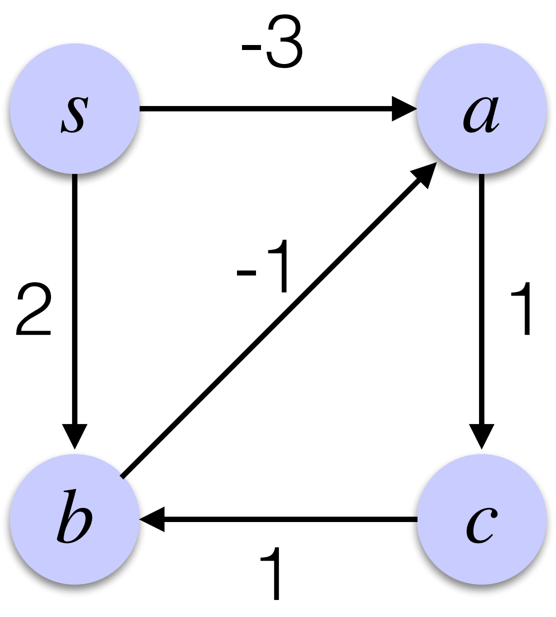
- $D[v,1] = \min\{D[v,0], \min_{u,v \in E} \{D[u,0] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3		
b	inf	2		
c	inf			



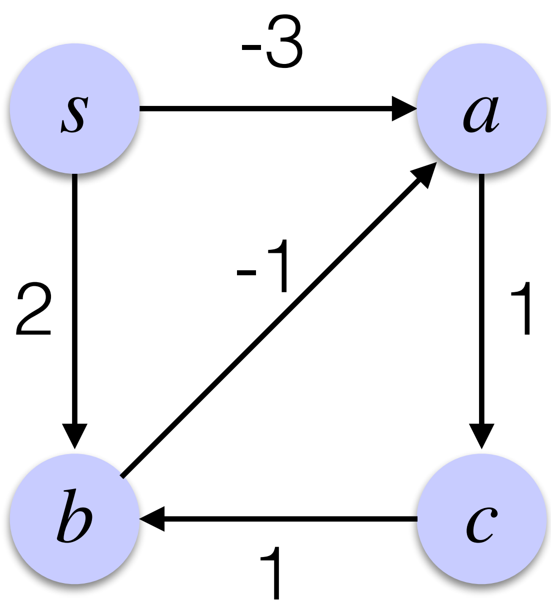
- $D[v,1] = \min\{D[v,0], \min_{u,v \in E} \{D[u,0] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3		
b	inf	2		
c	inf	inf		



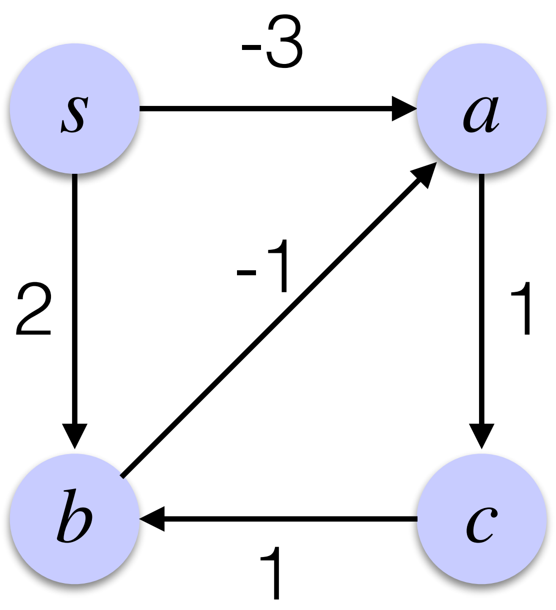
- $D[v,2] = \min\{D[v,1], \min_{u,v \in E} \{D[u,1] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3		
b	inf	2		
c	inf	inf		



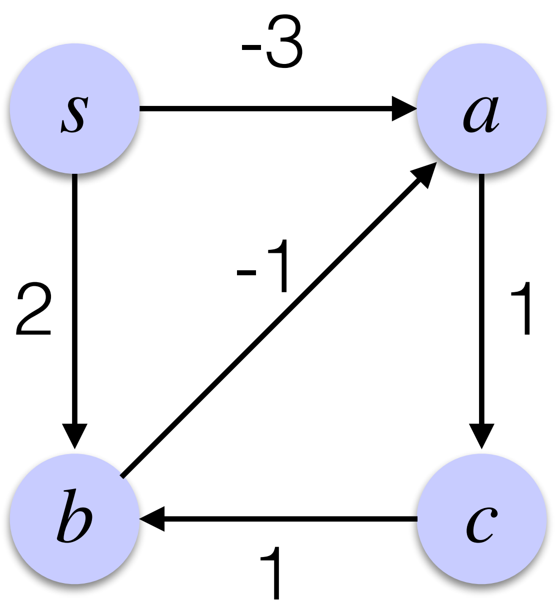
- $D[v,2] = \min\{D[v,1], \min_{u,v \in E} \{D[u,1] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	
b	inf	2		
c	inf	inf		



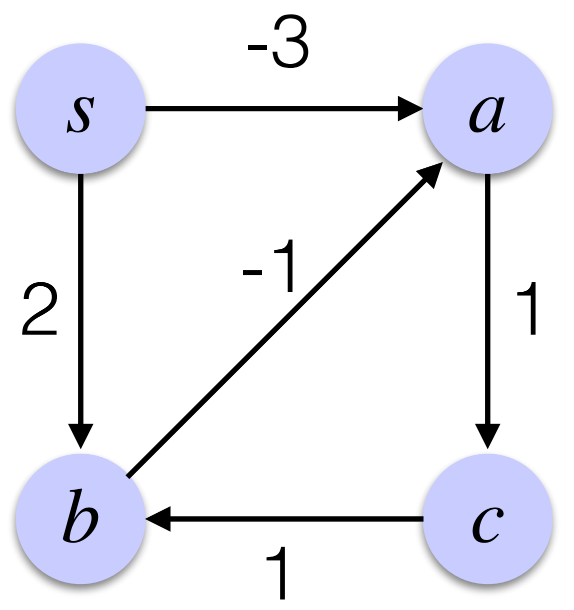
- $D[v,2] = \min\{D[v,1], \min_{u,v \in E} \{D[u,1] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	
b	inf	2	2	
c	inf	inf		



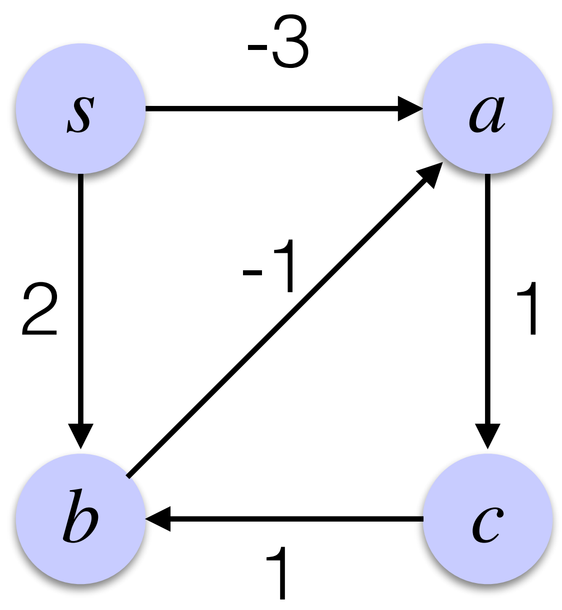
- $D[v,2] = \min\{D[v,1], \min_{u,v \in E} \{D[u,1] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	
b	inf	2	2	
c	inf	inf	-2	



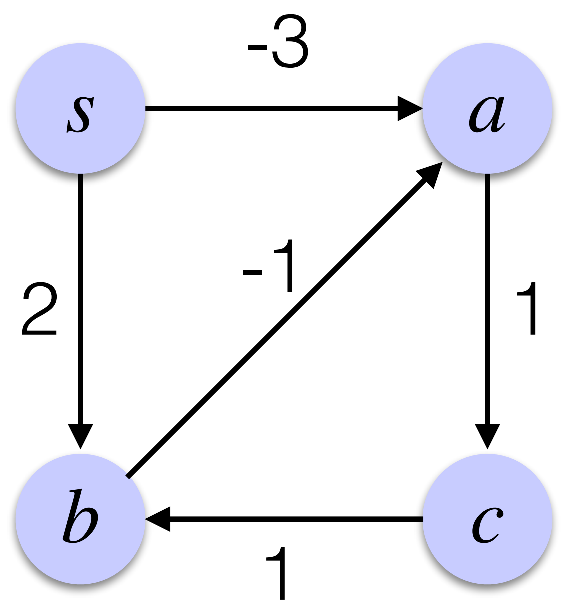
- $D[v,3] = \min\{D[v,2], \min_{u,v \in E} \{D[u,2] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	-3
b	inf	2	2	
c	inf	inf	-2	



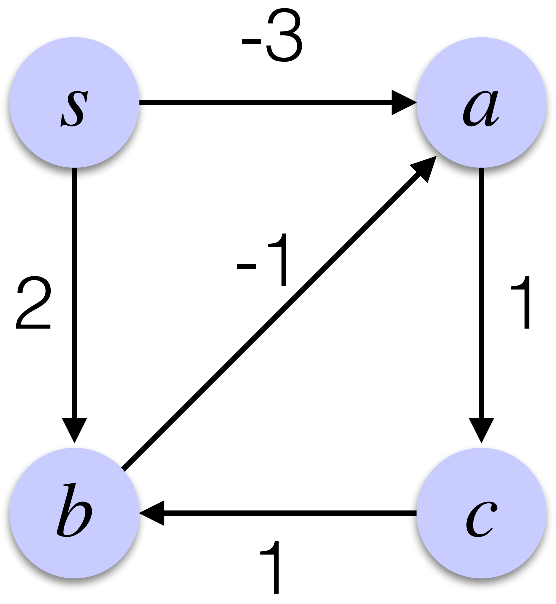
- $D[v,3] = \min\{D[v,2], \min_{u,v \in E}\{D[u,2] + w_{uv}\}$

	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	-3
b	inf	2	2	-1
c	inf	inf	-2	



- $D[v,3] = \min\{D[v,2], \min_{u,v \in E} \{D[u,2] + w_{uv}\}$

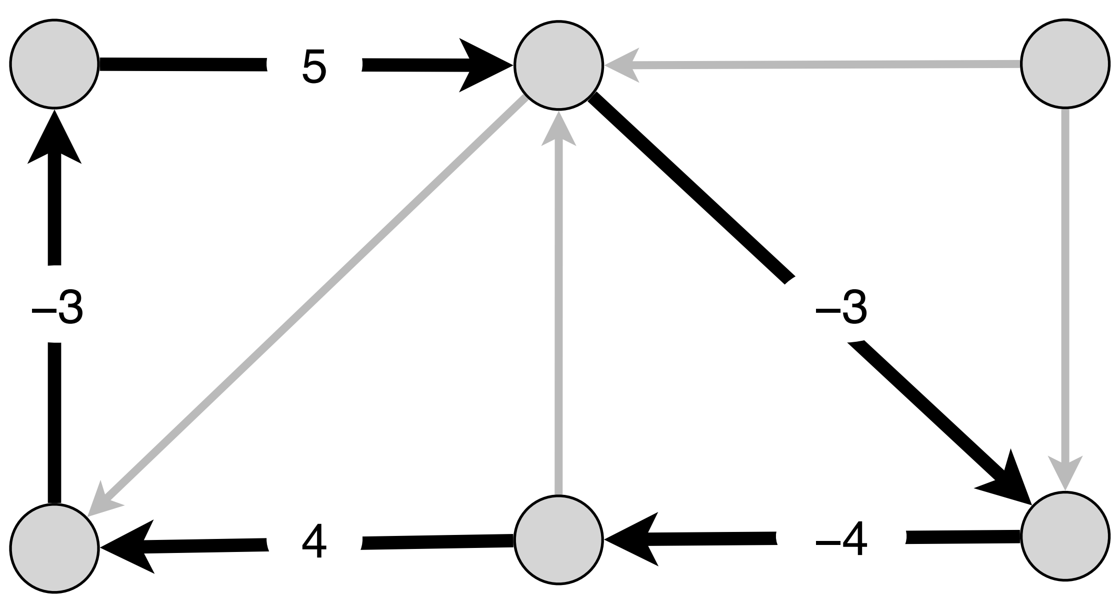
	0	1	2	3
s	0	0	0	0
a	inf	-3	-3	-3
b	inf	2	2	-1
c	inf	inf	-2	-2



Dynamic Programming Shortest Path: Detecting a Negative Cycle

Negative Cycle

- **Definition**. A negative cycle is a directed cycle C such that the sum of all the edge weights in C is less than zero
- **Claim.** If a path from s to some node v contains a negative cycle, then there does not exist a shortest path from s to v.



a negative cycle W : $\ell(W) = \sum_{e \in W} \ell_e < 0$

Detecting a Negative Cycle

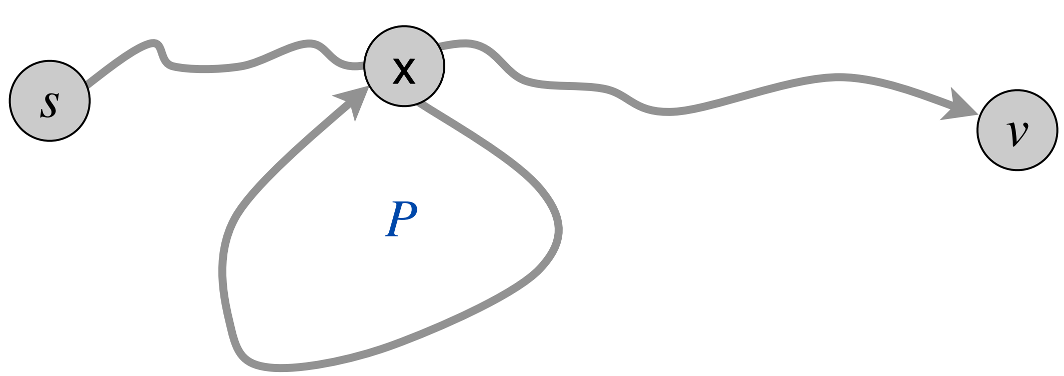
- **Question.** Given a directed graph $G = (V, E)$ with edge-weights w_e (can be negative), determine if G contains a negative cycle.
- Now, we don't have a specific source node given to us
- Let's change this problem a little bit
- **Problem**. Given G and source s, find if there is negative cycle on a $s \rightsquigarrow v$ path for any node v.

Detecting a Negative Cycle

- **Problem**. Given G and source s, find if there is negative cycle on a $s \rightsquigarrow v$ path for any node v.
- $D[v, i]$ is the cost of the shortest path from s to v of length at most i
- Suppose there is a negative cycle on a $s \rightsquigarrow v$ path
    - Then $\lim_{i \to \infty} D[v, i] = -\infty$
- If $D[v, n] = D[v, n-1]$ for every node v then G has no negative cycle!
    - Table values converge, no further improvements possible
    - OK, so if $D[v, n] = D[v, n-1]$ for all v we have no negative cycle. Is this all we need to check? (Can we prove if and only if?)

Detecting a Negative Cycle

- **Lemma.** If $D[v, n] < D[v, n-1]$ then any shortest $s \rightsquigarrow v$ path contains a negative cycle.
- **Proof**. [By contradiction] Suppose G does not contain a negative cycle
- Since $D[v, n] < D[v, n-1]$, the shortest $s \rightsquigarrow v$ path that caused this update has exactly n edges
- By pigeonhole principle, path must contain a repeated node, let the cycle between two successive visits to the node be P
- If P has non-negative weight, removing it would give us a shortest path with less than n edges $\Rightarrow\!\Leftarrow$



Analysis: First Attempt

- Now we know how to detect negative cycles on a shortest path from s to some node v.
- How do we detect a negative cycle anywhere in G?
- Do the above for each $s \in V$
- Running time?
  - $O(nm \cdot n) = O(n^2 m)$
  - Can we improve this?

Problem Reduction

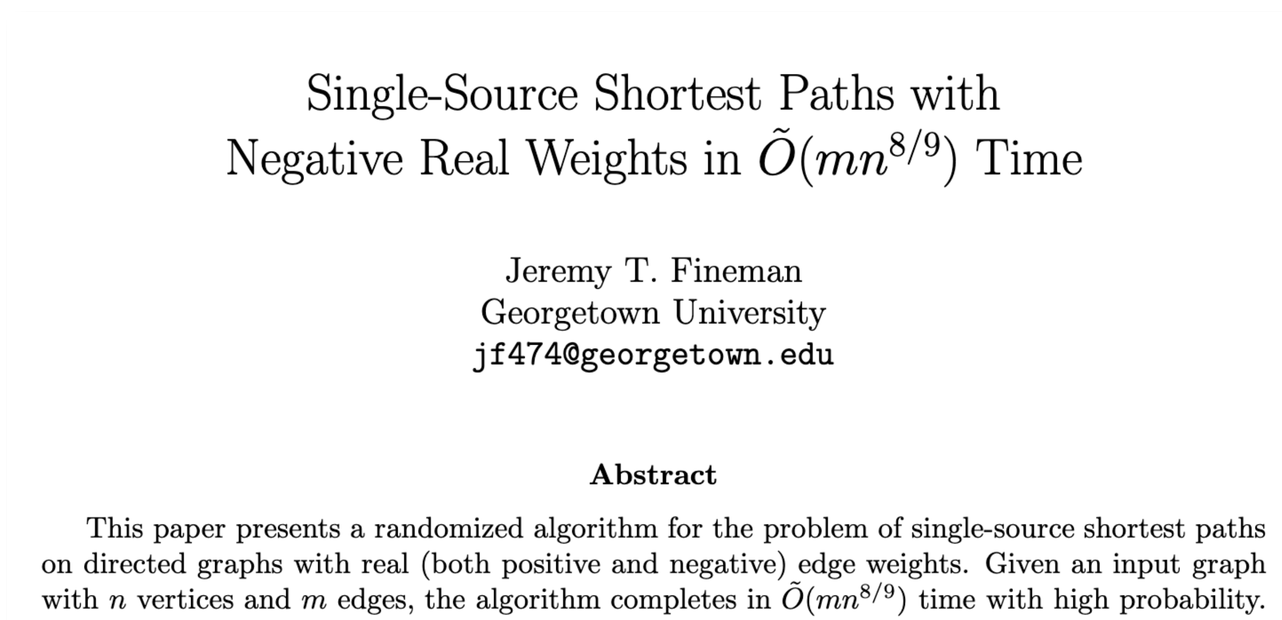
- Now we know how to detect negative cycles on a shortest path from s to some node v.
- How do we detect a negative cycle anywhere in G?
- **Reduction**. Given graph G, add a source s and connect it to all vertices in G with edge weight 0. Let the new graph be G'
- **Claim**. G has a negative cycle iff G' has a negative cycle from s to some node v.
- **Proof**. $\Rightarrow$ If G has a negative cycle, then this cycle lies on the shortest path from s to a node on the cycle in G'
- $\Leftarrow$ If G' has a negative cycle on a shortest path from s to some node, then that node is on a negative cycle in G

Problem Reduction

- Running time is now $O(nm)$ rather than $O(n^2m)$
- Idea: our original algorithm was for a slightly different problem than what we wanted. Rather than running it over and over, we **changed the input** and ran it once
  - Gave us the answer for the final problem
  - We'll see many more reductions in part 3 of the course

Bellman-Ford Fun Facts

- Can we improve on $O(nm)$ for single source shortest paths with negative edges?
- Open problem since invention in 1956
- [Fineman 2024]: $O(n^{8/9}m)$ algorithm
  - Uses a very clever and complicated *reduction* to Dijkstra's algorithm
  - [Huang Jin Quanrud 2025]: $O(n^{4/5}m)$ algorithm



Single-Source Shortest Paths with
Negative Real Weights in $\tilde{O}(mn^{8/9})$ Time

Jeremy T. Fineman
Georgetown University
jf474@georgetown.edu

Abstract

This paper presents a randomized algorithm for the problem of single-source shortest paths on directed graphs with real (both positive and negative) edge weights. Given an input graph with n vertices and m edges, the algorithm completes in $\tilde{O}(mn^{8/9})$ time with high probability.

DP Coding Example

Coding up DP

- We have talked mostly about “filling out a recipe” and “what does the table look like”
- These are real techniques to solve algorithmic problems using computers
- Let’s look at how one might code these up
- Using very basic python

Reminder: Recipe for LIS

- **Subproblem.** $L[i]$ stores longest subsequence ending at i
- **Recurrence.** $L[i] = 1 + \max_{m \in M} L[m]$ where $M = \{j \mid j < i \text{ and } A[j] < A[i]\}$
- **Base case**. $L[0] = 1$
- **Final answer.** $\max_i L[i]$
- **Memoization data structure.** L is an array of length n
- **Evaluation order.** Increasing order of i

- How to recover solution: the m we chose is the second-to-last element in the solution. Store all m in an array B, and walk backwards through B to recover solution

Introduction to Network Flows

Story So Far

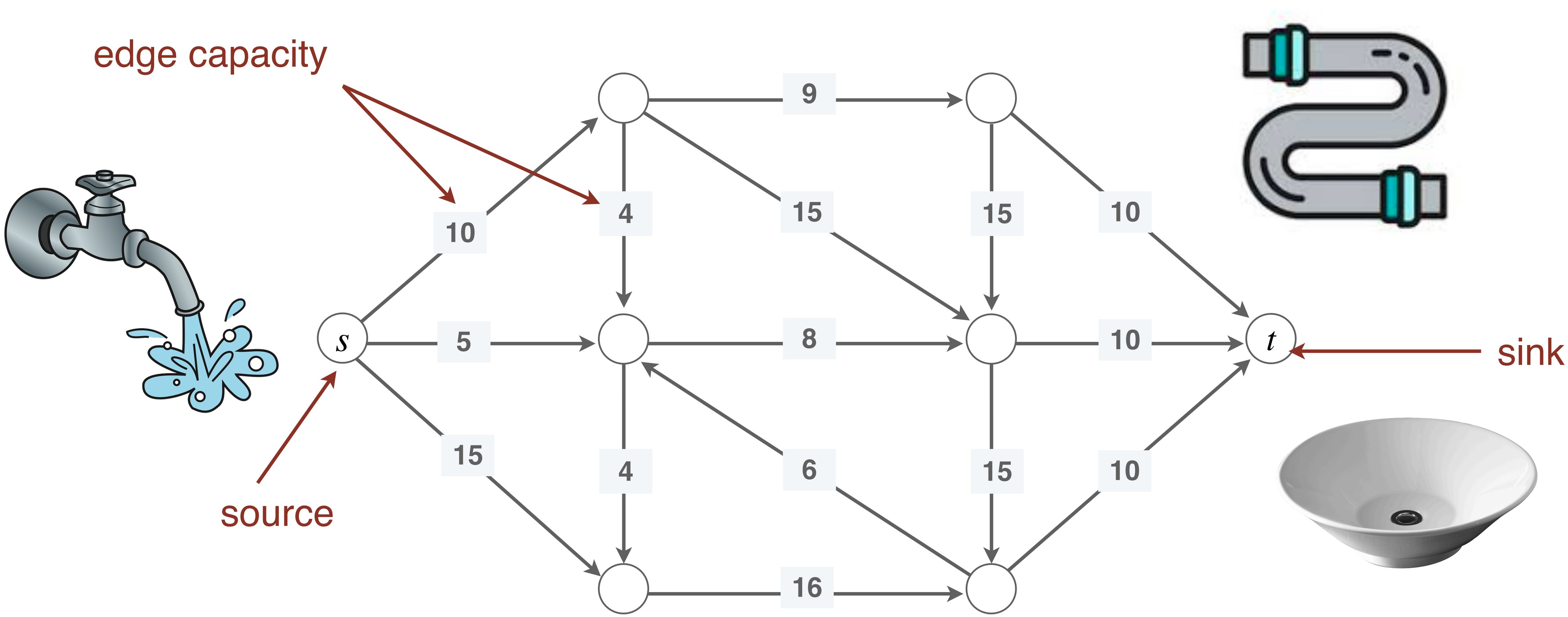
- Algorithmic design paradigms:
  - **Greedy**: simplest to design but works only for certain limited class of optimization problems
    - A good starting point for most problems but rarely optimal
  - **Divide and Conquer**
    - Solving a problem by breaking it down into smaller subproblems and recursing
  - **Dynamic programming**
    - Recursion with memoization: avoiding repeated work
    - Trading off space for time

Network Flows

- Graph-based problem; looks like a lot of what we learned in part 1
- Soon, we'll use what we learn about network flows to solve much more general problems
- Problems where you **revisit*** (and improve) past solutions
- Solve problems that even dynamic programming can't* solve!
- Restricted case of Linear/Convex Programming; "algorithmic power tools"

What's a Flow Network?

- A flow network is a directed graph $G = (V, E)$ with a
  - A **source** is a vertex s with in degree 0
  - A **sink** is a vertex t with out degree 0
  - Each edge $e \in E$ has **edge capacity** $c(e) > 0$



Visualize

Assumptions

- Assume that each node v is on some s-t path, that is, $s \rightsquigarrow v \rightsquigarrow t$ exists, for any vertex $v \in V$
    - Implies G is connected and $m \geq n - 1$
- Assume **capacities are integers**
    - Will revisit this assumption and what happens if not
- Directed edge (u, v) written as $u \rightarrow v$
- For simplifying expositions, we will sometimes write $c(u \rightarrow v) = 0$ when $(u, v) \notin E$

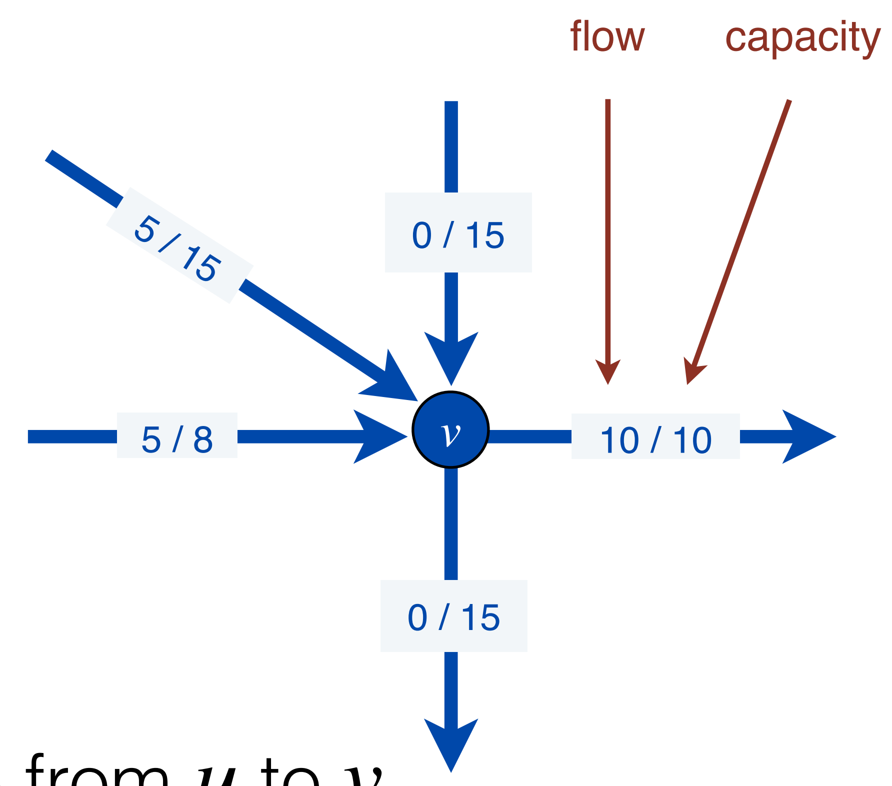
What's a Flow?

- Given a flow network, an (s, t)**-flow** or just **flow** (if source s and sink t are clear from context) $f : E \to \mathbb{Z}^+$ satisfies the following two constraints:

- **[Flow conservation]** $f_{in}(v) = f_{out}(v)$, for $v \neq s, t$ where

$$f_{in}(v) = \sum_{u} f(u \to v)$$

$$f_{out}(v) = \sum_{w} f(v \to w)$$

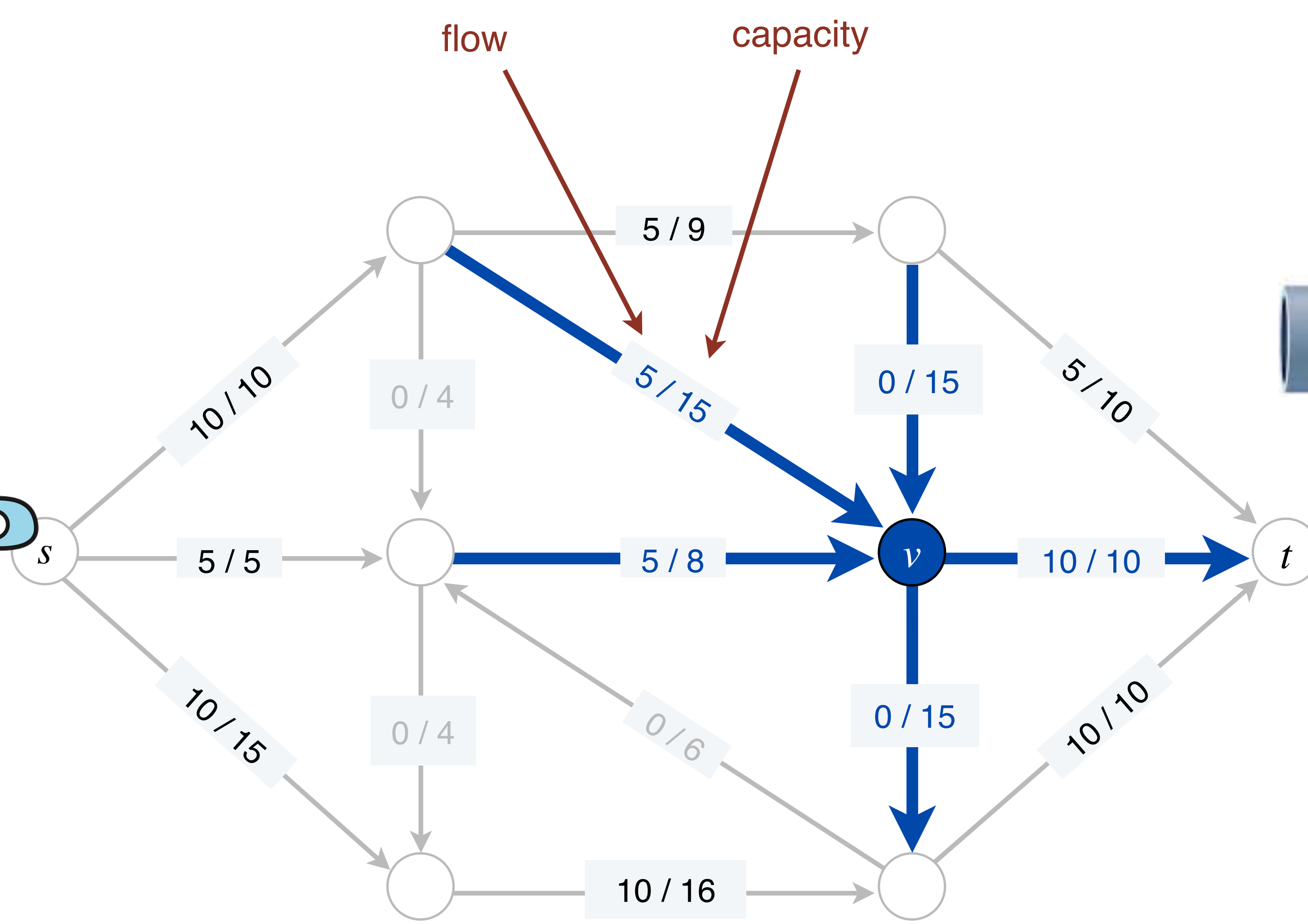


- To simplify, $f(u \to v) = 0$ if there is no edge from u to v

Feasible Flow

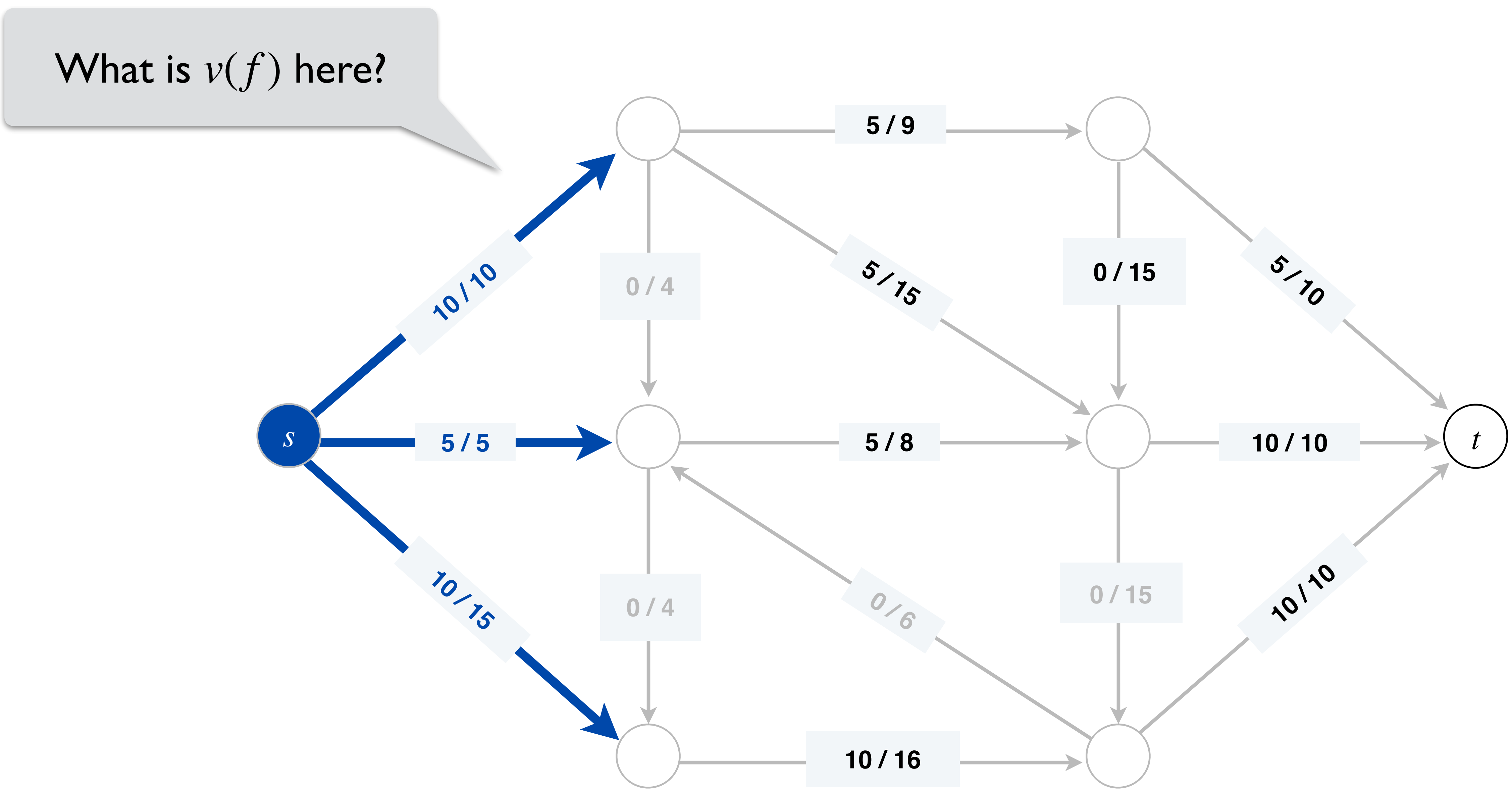
- And second, a feasible flow must satisfy the capacity constraints of the network, that is,

[Capacity constraint] for each $e \in E$, $0 \leq f(e) \leq c(e)$



Value of a Flow

- **Definition.** The **value** of a flow f, written $v(f)$, is $f_{out}(s)$.

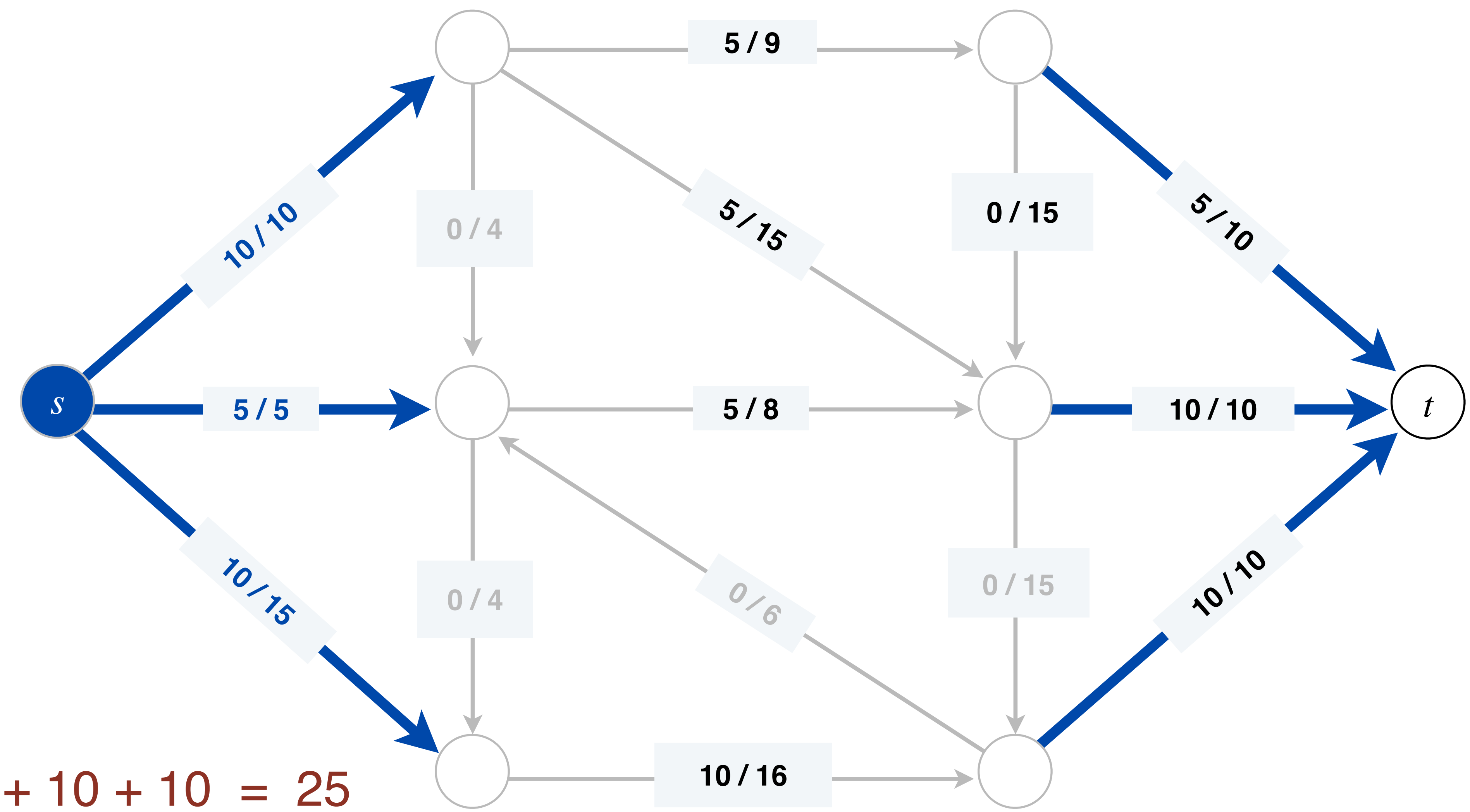


$v(f) = 5 + 10 + 10 = 25$

Value of a Flow

- **Definition.** The **value** of a flow f, written $v(f)$, is $f_{out}(s)$.
- **Lemma**. $f_{out}(s) = f_{in}(t)$

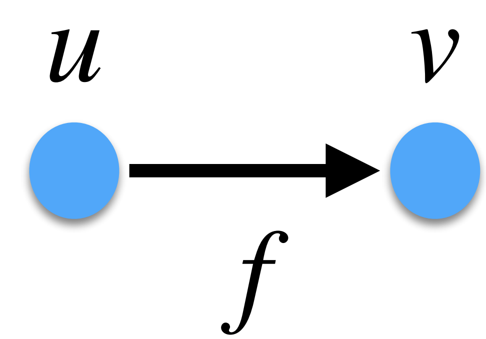
Intuitively, why do you think this is true?



value = 5 + 10 + 10 = 25

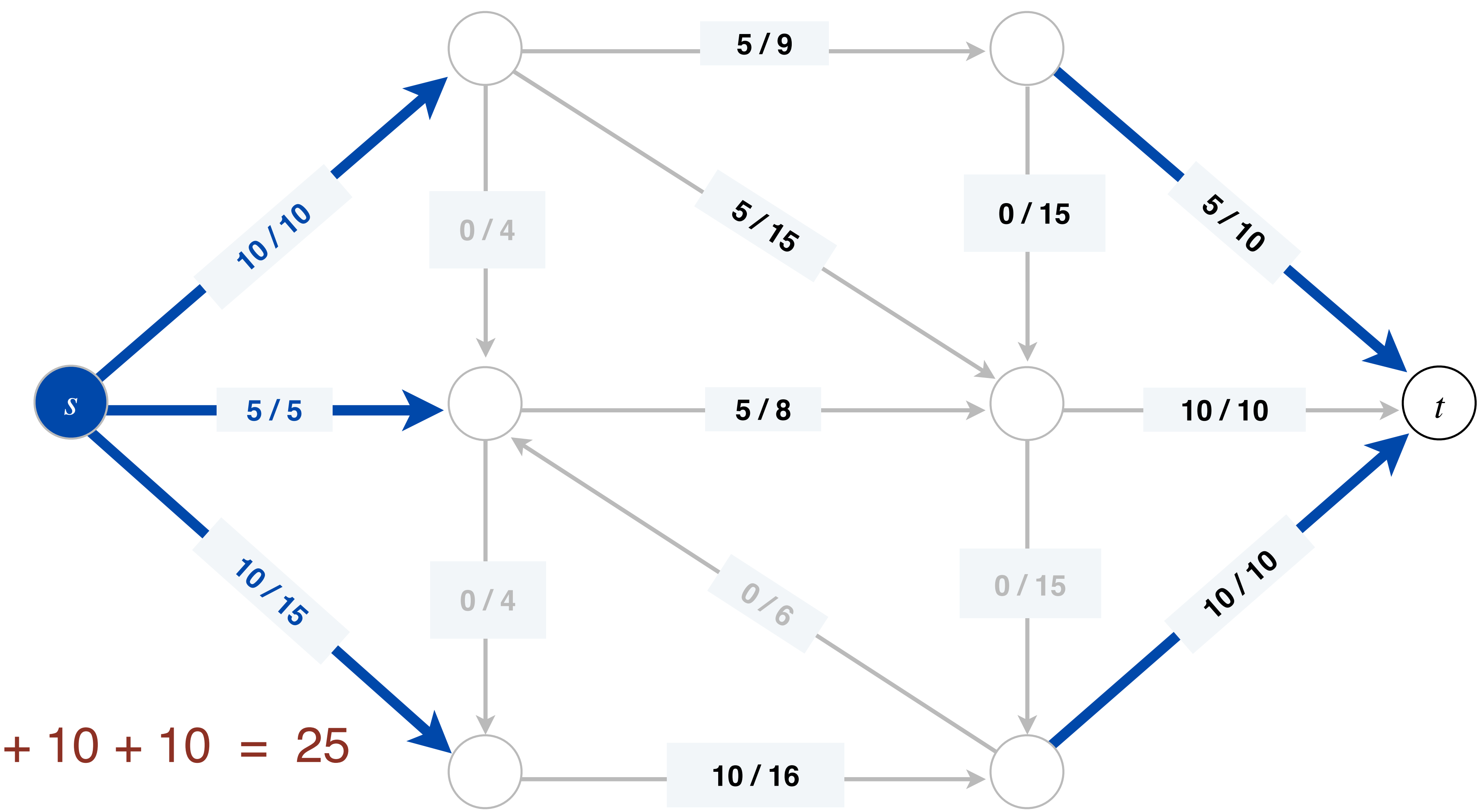
Value of a Flow

- **Lemma**. $f_{out}(s) = f_{in}(t)$
- **Proof**. Let $f(E) = \sum_{e \in E} f(e)$
  - Then, $\sum_{v \in V} f_{in}(v) = f(E) = \sum_{v \in V} f_{out}(v)$
  - For every $v \neq s, t$ flow conversation implies $f_{in}(v) = f_{out}(v)$
  - Thus all terms cancel out on both sides except
    $f_{in}(s) + f_{in}(t) = f_{out}(s) + f_{out}(t)$
  - But $f_{in}(s) = f_{out}(t) = 0$ ∎



Value of a Flow

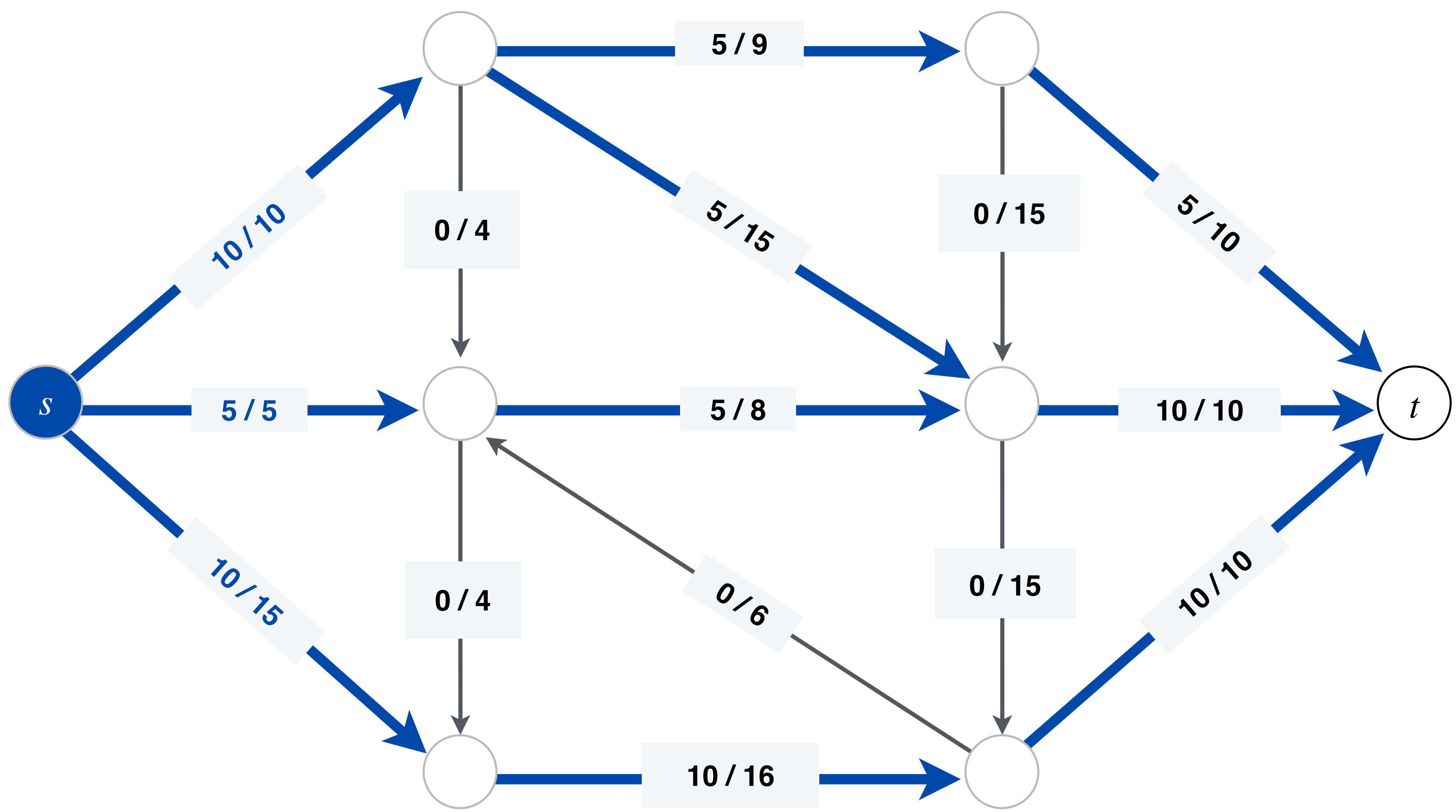
- **Lemma**. $f_{out}(s) = f_{in}(t)$
- **Corollary.** $v(f) = f_{in}(t)$.



value = 5 + 10 + 10 = 25

Max-Flow Problem

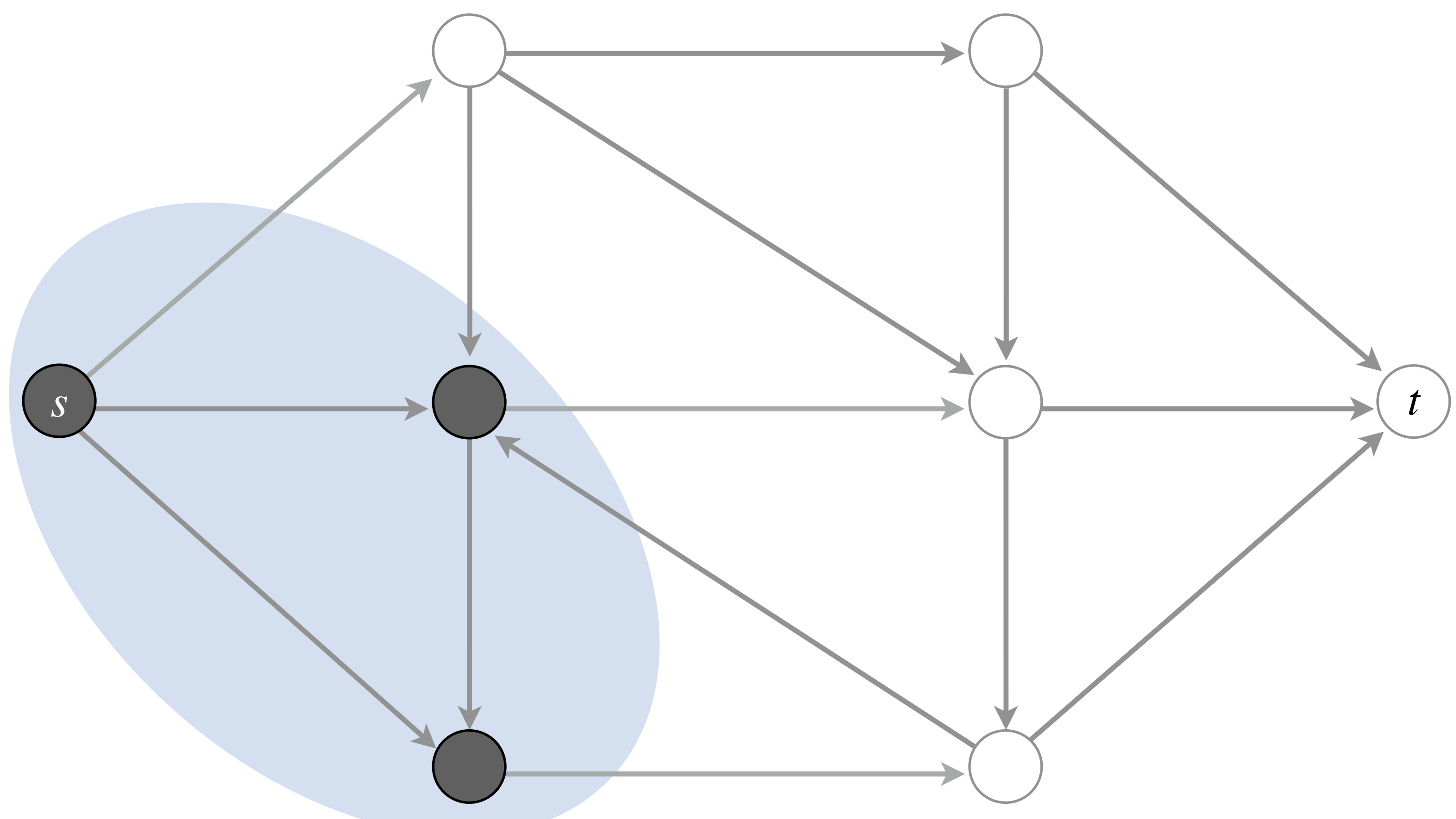
- **Problem**. Given an s-t flow network, find a feasible s-t flow of **maximum** value.



Minimum Cut Problem

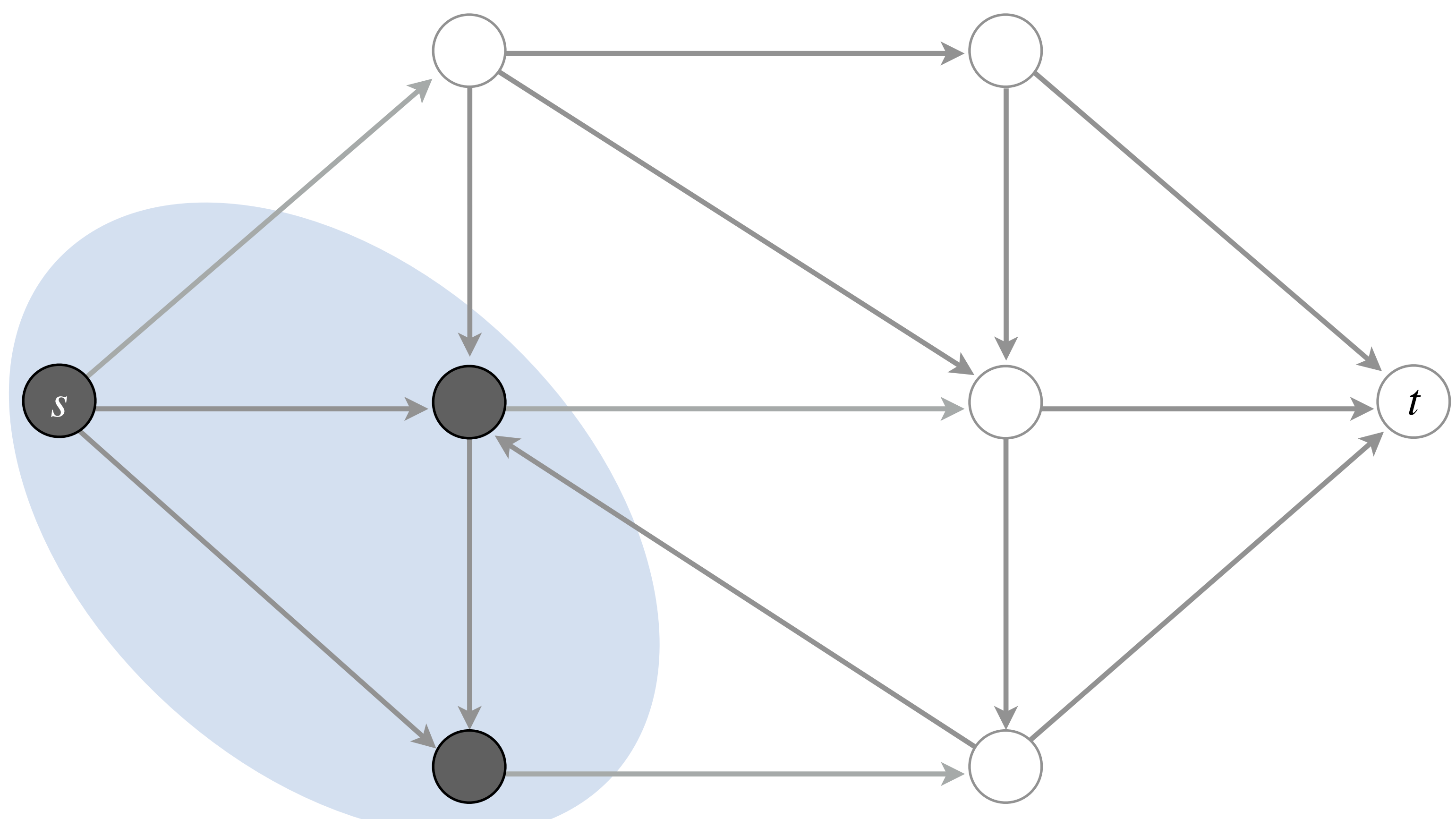
Cuts are Back!

- Cuts in graphs played a lead role when we were designing algorithms for MSTs
- What is the definition of a cut?



Cuts in Flow Networks

- **Recall**. A cut (S, T) in a graph is a partition of vertices such that $S \cup T = V$, $S \cap T = \varnothing$ and S, T are non-empty.
- **Definition**. An (s, t)-*cut* is a cut (S, T) s.t. $s \in S$ and $t \in T$.



Cut Capacity

- **Recall**. A cut (S, T) in a graph is a partition of vertices such that $S \cup T = V$, $S \cap T = \emptyset$ and S, T are non-empty.
- **Definition**. An (s, t)-*cut* is a cut (S, T) s.t. $s \in S$ and $t \in T$.
- **Capacity** of a (s, t)-cut (S, T) is the sum of the capacities of edges leaving S:
  - $$c(S, T) = \sum_{v \in S, w \in T} c(v \to w)$$

Quick Quiz

Question. What is the capacity of the s-t cut given by the grey and white nodes?

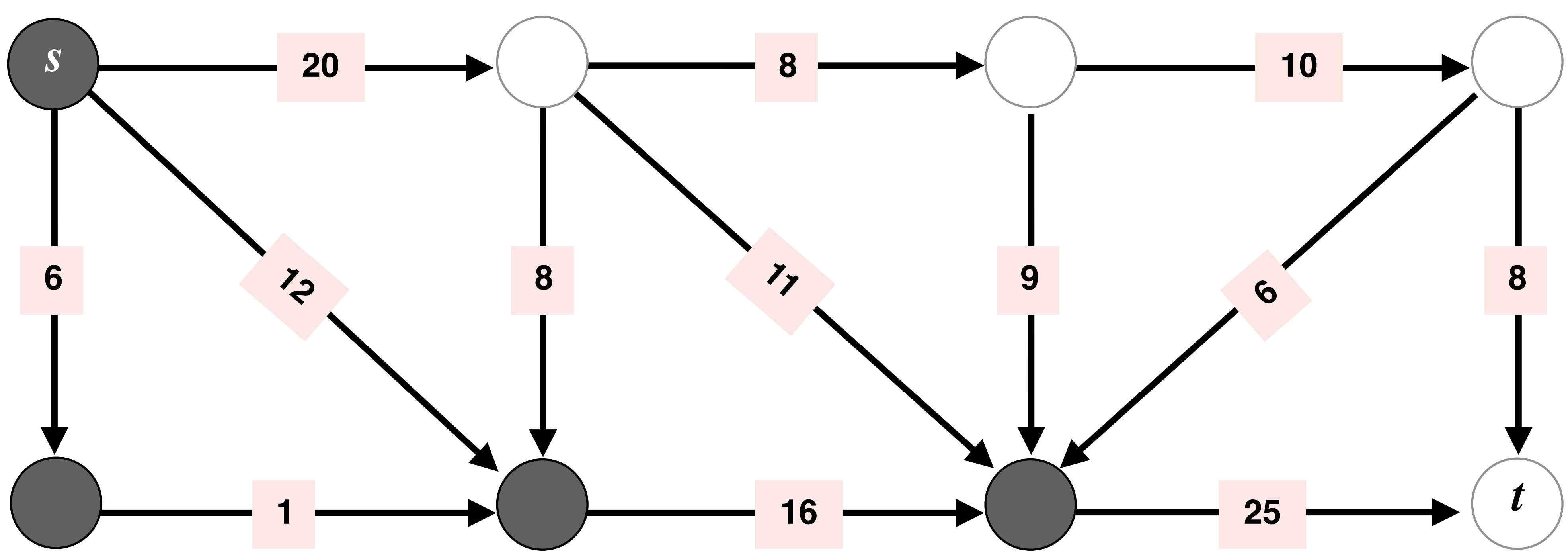
$$c(S,T) = \sum_{v \in S, w \in T} c(v \to w)$$

A. 11 (20 + 25 – 8 – 11 – 9 – 6)

B. 34 (8 + 11 + 9 + 6)

C. 45 (20 + 25)

D. 79 (20 + 25 + 8 + 11 + 9 + 6)



Quick Quiz

Question. What is the capacity of the s-t cut given by the grey and white nodes?

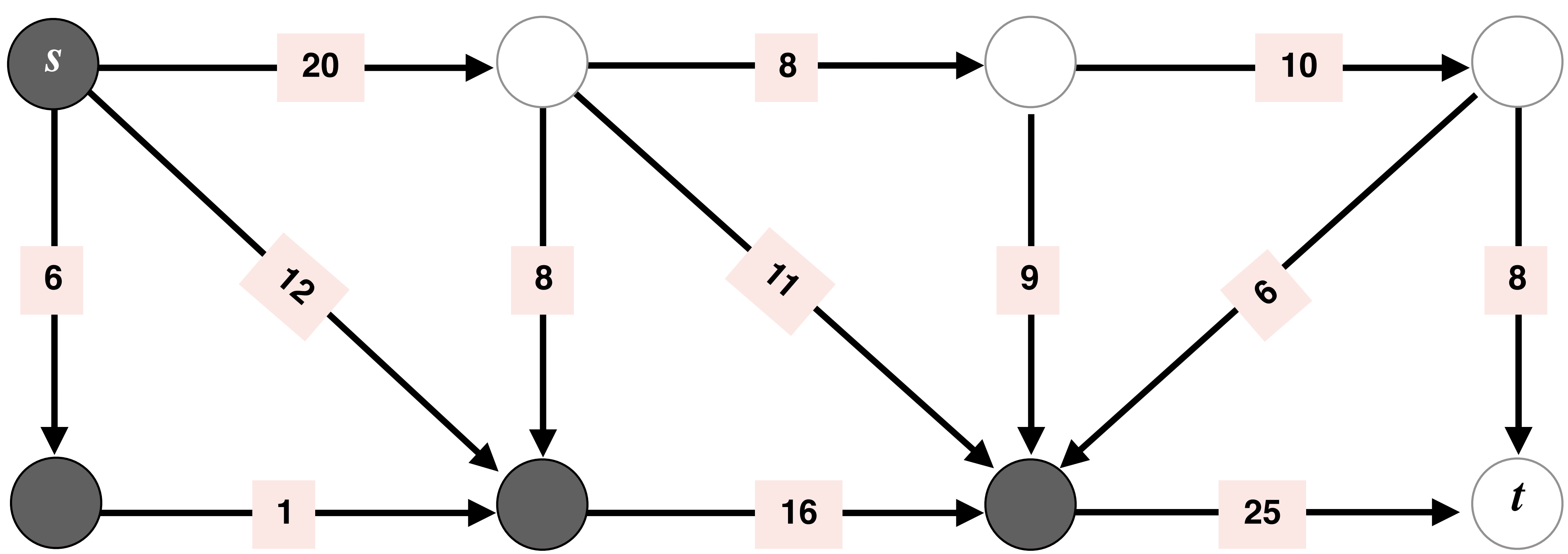
$$c(S, T) = \sum_{v \in S, w \in T} c(v \to w)$$

A. 11 (20 + 25 – 8 – 11 – 9 – 6)

B. 34 (8 + 11 + 9 + 6)

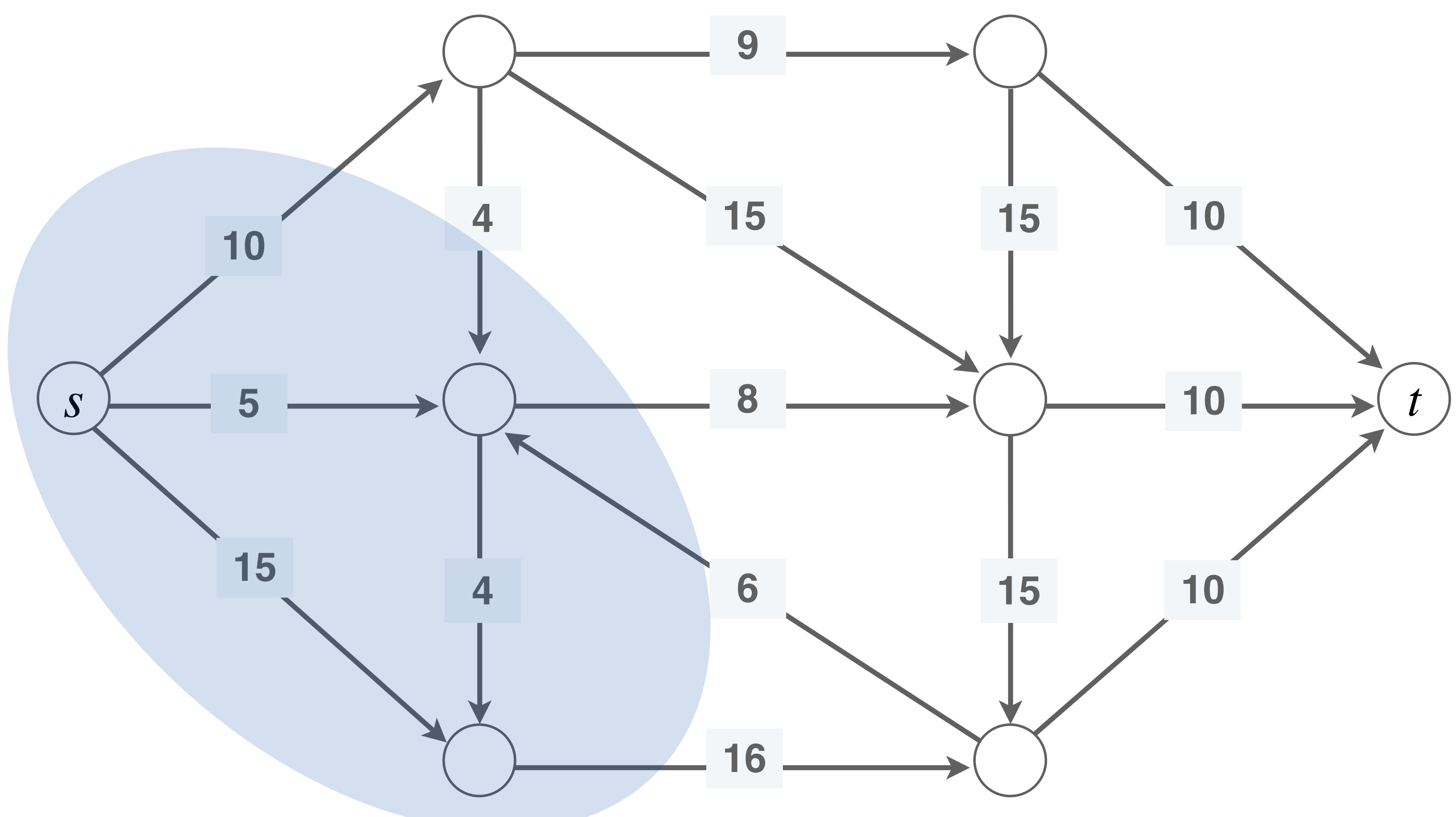
C. 45 (20 + 25)

D. 79 (20 + 25 + 8 + 11 + 9 + 6)



Min Cut Problem

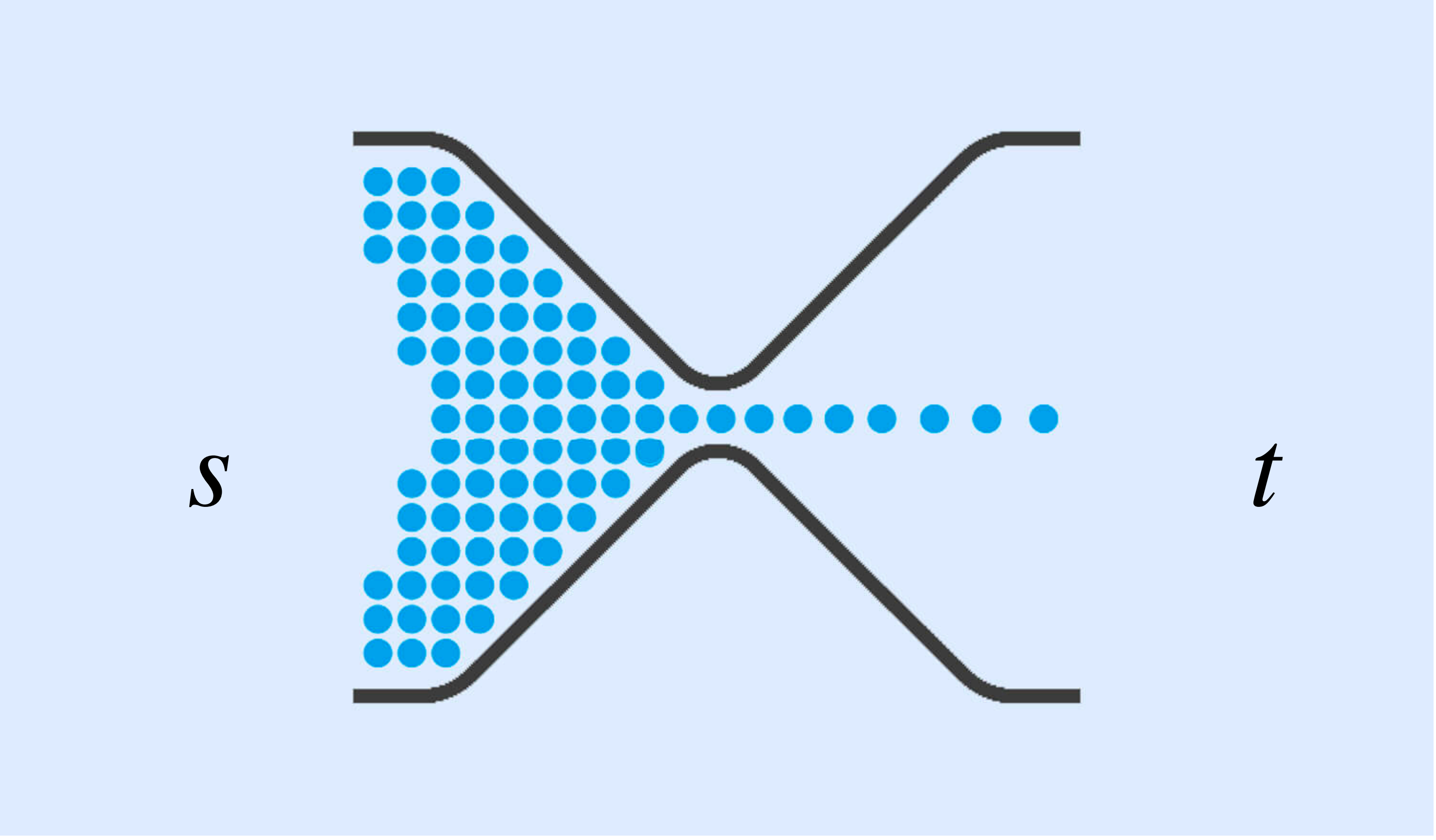
- **Problem**. Given an *s*-*t* flow network, find an *s*-*t* cut of **minimum** capacity.



Relationship between Flows and Cuts

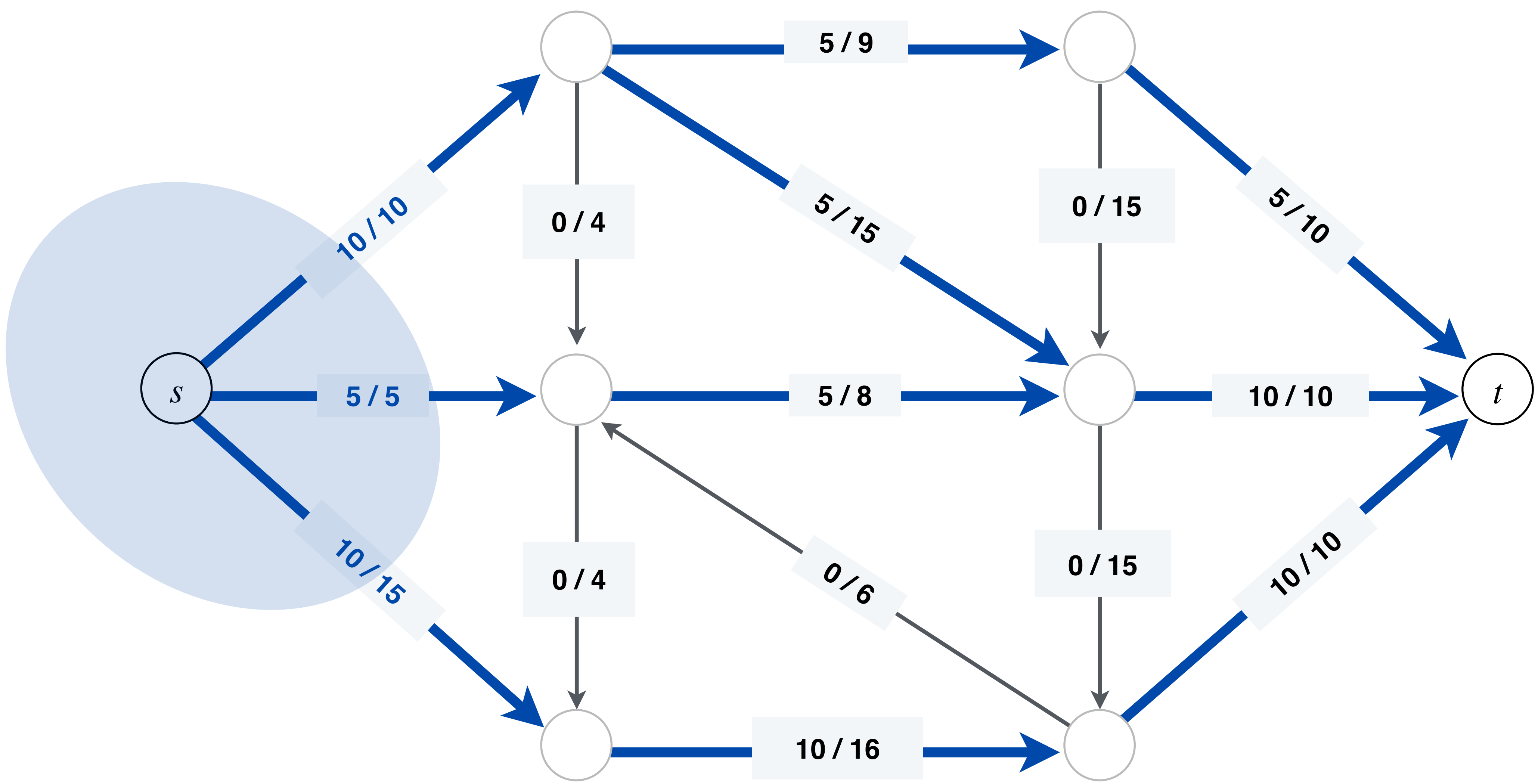
Flows and Cuts

- Cuts represent "**bottlenecks**" in a flow network
- For any cut, our flow needs to “get out” of that cut on its route from s to t
- Let us formalize this intuition



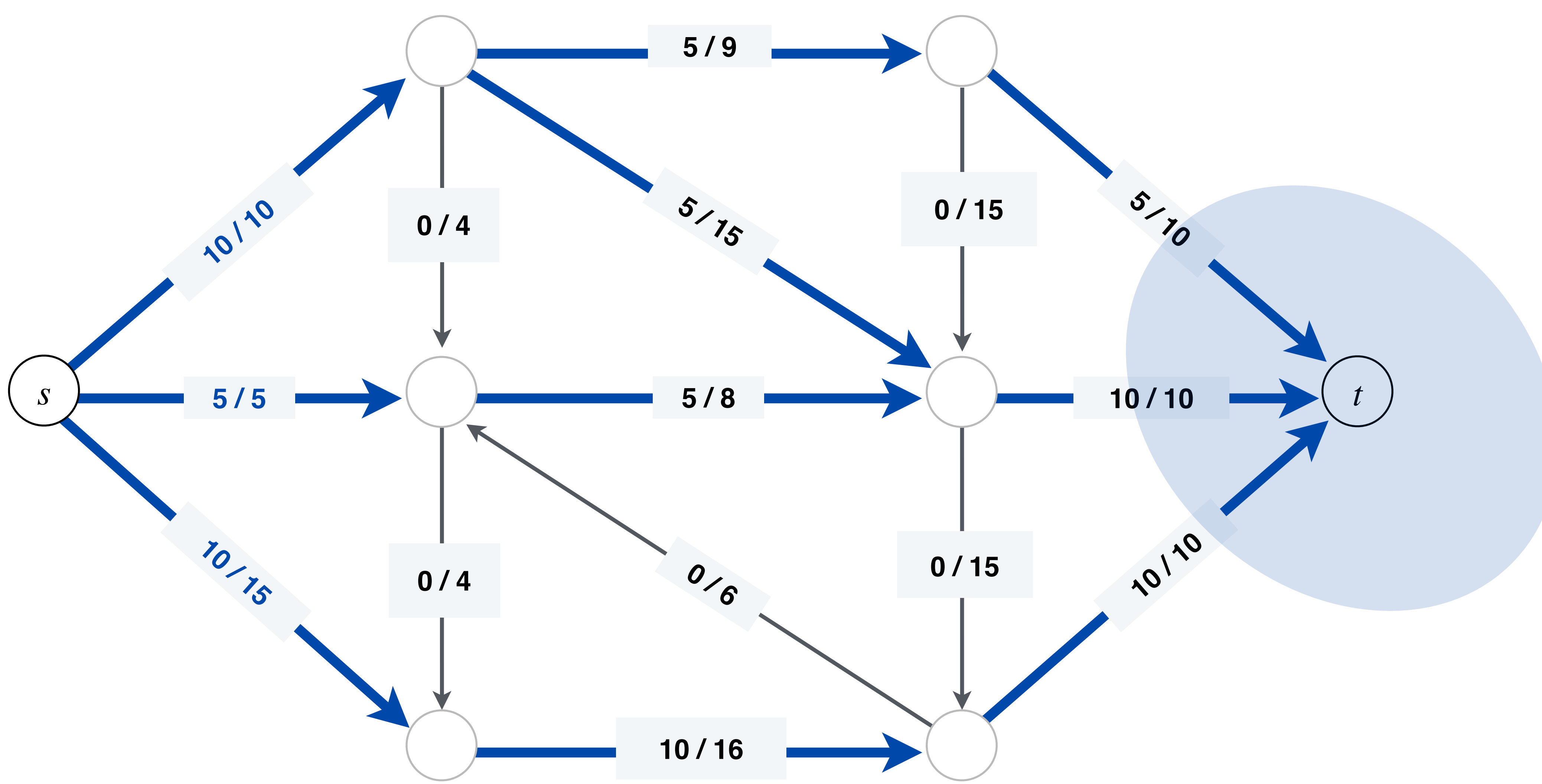
Flows and Cuts

- **Claim**. Let f be **any** s-t flow and (S, T) be **any** s-t cut then $v(f) \leq c(S, T)$
- There are two s-t cuts for which this is easy to see, which ones?



Flows and Cuts

- **Claim**. Let f be **any** s-t flow and (S, T) be **any** s-t cut then $v(f) \leq c(S, T)$
- There are two s-t cuts for which this is easy to see, which ones?



Flows and Cuts

- To prove this for any cut, we first relate the flow value in a network to the net flow leaving a cut
- **Lemma**. For any feasible (s, t)-flow f on $G = (V, E)$ and any (s, t)-cut , $v(f) = f_{out}(S) - f_{in}(S)$, where
  - $f_{out}(S) = \sum_{v \in S, w \in T} f(v \to w)$ (sum of flow 'leaving' S)
  - $f_{in}(S) = \sum_{v \in S, w \in T} f(w \to v)$ (sum of flow 'entering' S)
  - Note: $f_{out}(S) = f_{in}(T)$ and $f_{in}(S) = f_{out}(T)$

Flows and Cuts

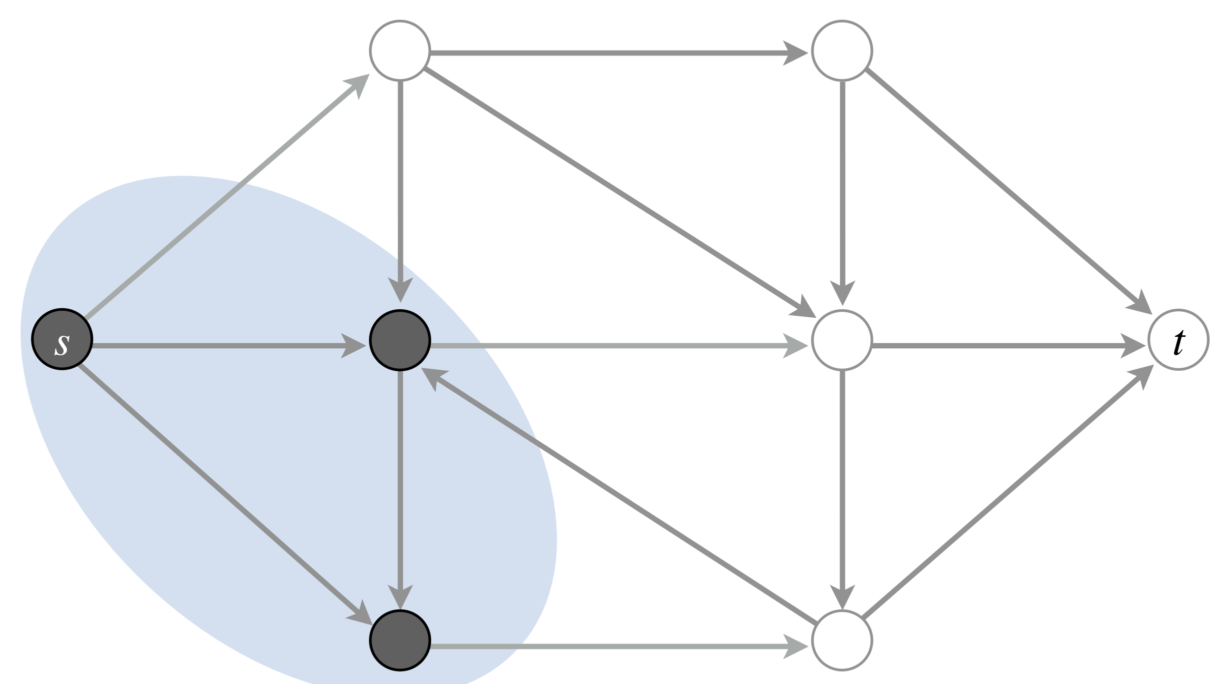
Proof. $f_{out}(S) - f_{in}(S)$

$$= \sum_{v\in S, w\in T} f(v \to w) - \sum_{v\in S, u\in T} f(u \to v) \quad \text{[by definition]}$$

Adding zero terms

$$= \left[\sum_{v,w\in S} f(v \to w) - \sum_{v,u\in S} f(u \to v) \right] + \sum_{v\in S, w\in T} f(v \to w) - \sum_{v\in S, u\in T} f(u \to v)$$

These are the same sum: they sum the flow of all edges with both vertices in S



Flows and Cuts

Proof. $f_{out}(S) - f_{in}(S)$

$$= \left[\sum_{v,w \in S} f(v \to w) - \sum_{v,u \in S} f(u \to v) \right] + \sum_{v \in S, w \in T} f(v \to w) - \sum_{v \in S, u \in T} f(u \to v)$$

Rearranging terms

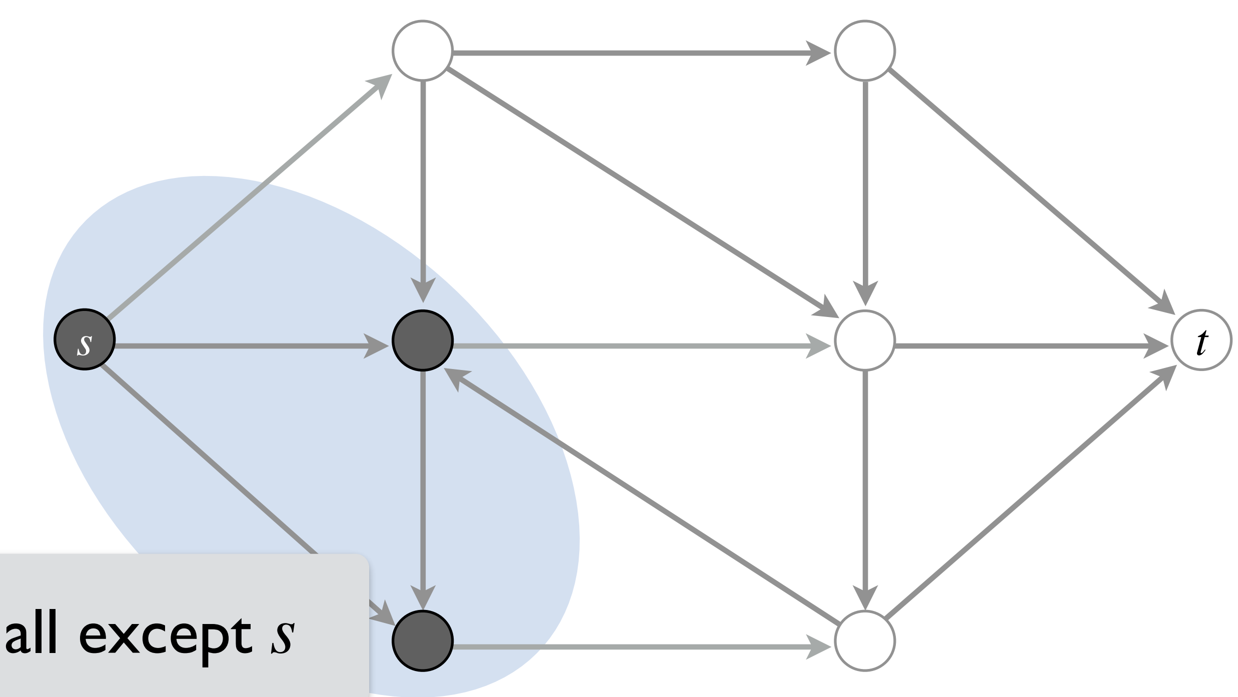
$$= \sum_{v,w \in S} f(v \to w) + \sum_{v \in S, w \in T} f(v \to w) - \sum_{v,u \in S} f(u \to v) - \sum_{v \in S, u \in T} f(u \to v)$$

$$= \sum_{v \in S} \left(\sum_{w} f(v \to w) - \sum_{u} f(u \to v) \right)$$

$$= \sum_{v \in S} f_{out}(v) - f_{in}(v)$$

Cancels out for all except s

$$= f_{out}(s) = v(f) \quad \blacksquare$$



Flows and Cuts

- We use this result to prove that the value of a flow cannot exceed the capacity of any cut in the network
- **Claim**. Let f be any s-t flow and (S, T) be any s-t cut then $v(f) \leq c(S, T)$
- **Proof.** $v(f) = f_{out}(S) - f_{in}(S)$

$$\leq f_{out}(S) = \sum_{v \in S, w \in T} f(v \to w)$$

$$\leq \sum_{v \in S, w \in T} c(v, w) = c(S, T)$$

When is $v(f) = c(S, T)$?

$f_{in}(S) = 0,\ f_{out}(S) = c(S, T)$

Max-Flow & Min-Cut

- Suppose the $c_{\min}$ is the capacity of the minimum cut in a network
- What can we say about the feasible flow we can send through it
    - cannot be more than $c_{\min}$
- In fact, whenever we find any s-t flow f and any s-t cut (S, T) such that, $v(f) = c(S, T)$ we can conclude that:
    - f is the maximum flow, and,
    - (S, T) is the minimum cut
- The question now is, given any flow network with min cut $c_{\min}$, is it always possible to route a feasible s-t flow f with $v(f) = c_{\min}$

Max-Flow Min-Cut Theorem

- A beautiful, powerful relationship between these two problems in given by the following theorem
- **Theorem**. Given any flow network G, there exists a feasible (s, t)-flow f and a (s, t)-cut (S, T) such that,

$$v(f) = c(S, T)$$

- Informally, in a flow network, the max-flow = min-cut
- This will guide our algorithm design for finding max flow
- (Will prove this theorem by construction in a bit—our algorithm will prove the theorem! (like with Gale-Shapley))

Network Flow History

- In 1950s, US military researchers Harris and Ross wrote a classified report about the rail network linking Soviet Union and Eastern Europe
    - Vertices were the geographic regions
    - Edges were railway links between the regions
    - Edge weights were the rate at which material could be shipped from one region to next
- Ross and Harris determined:
    - Maximum amount of stuff that could be moved from Russia to Europe **(max flow)**
    - Cheapest way to disrupt the network by removing rail links **(min cut)**

Network Flow History

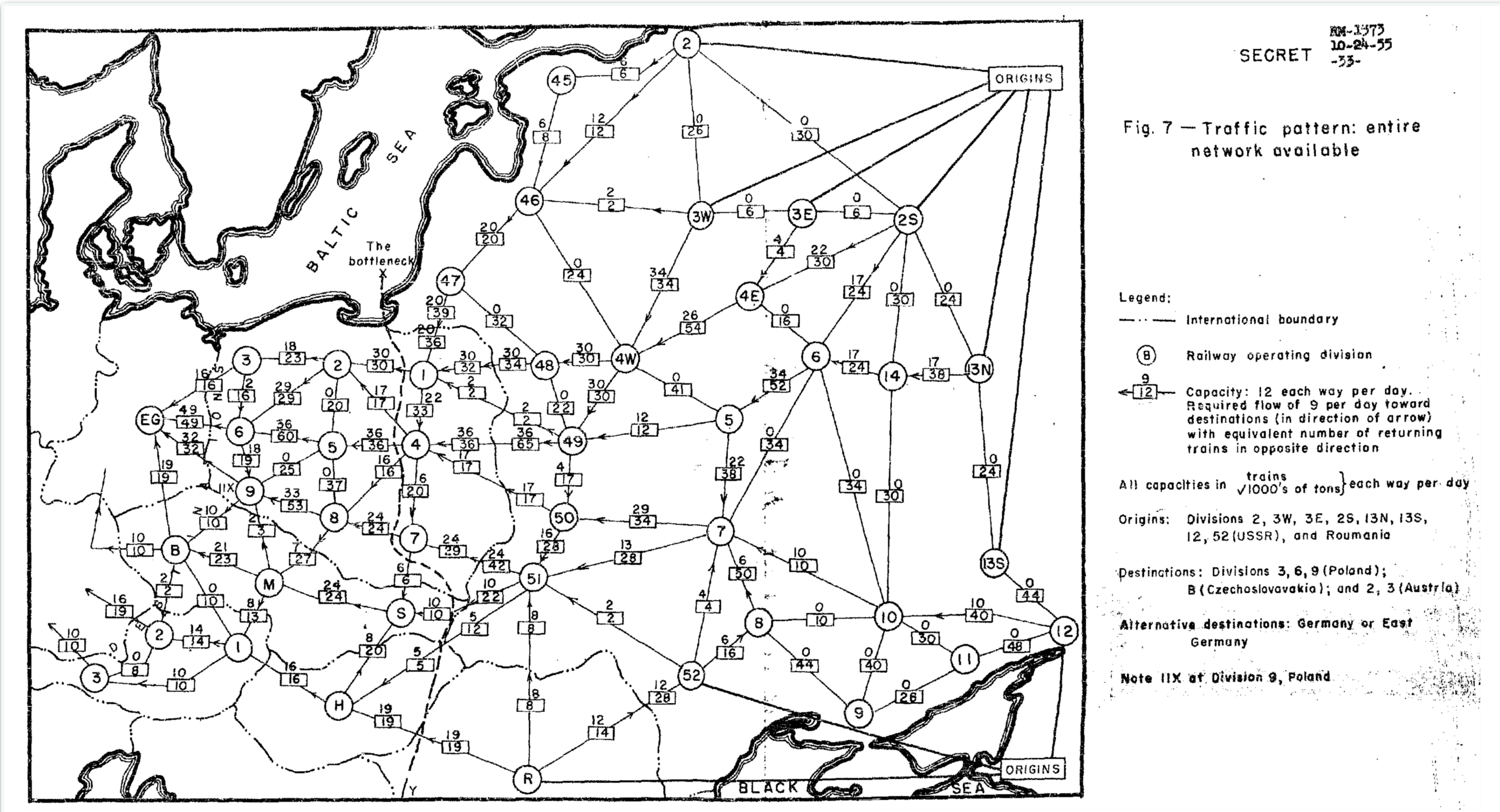


Image Credits: — Jeff Erickson's book and T[homas] E. Harris and F[rank] S. Ross. Fundamentals of a method for evaluating rail net capacities. The RAND Corporation, Research Memorandum RM-1517, October 24, 1955. United States Government work in the public domain. http://www.dtic.mil/dtic/tr/fulltext/u2/093458.pdf

Towards a Max-Flow Algorithm

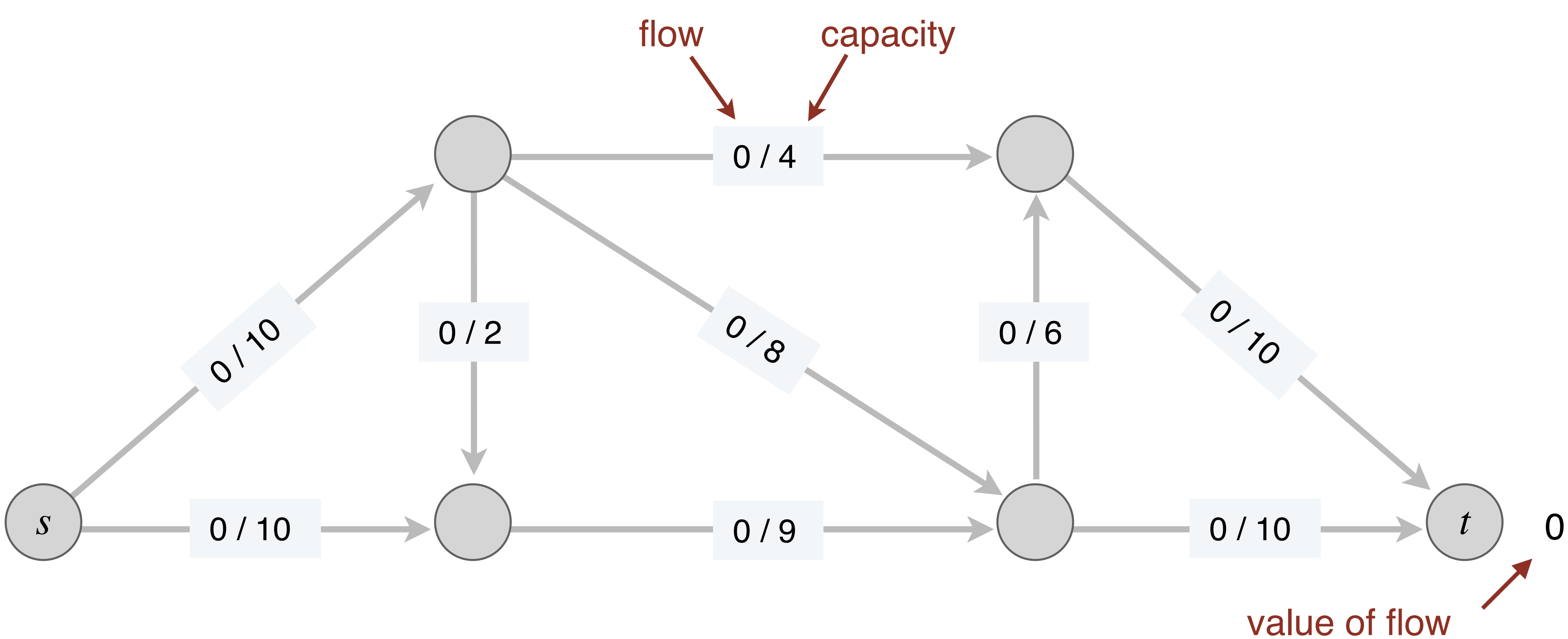
- Today: we will prove the max-flow min-cut theorem *constructively*
- We will design a max-flow algorithm and show that there is a s-t cut s.t. value of flow computed by algorithm = capacity of cut
- Let's start with a greedy approach
    - Push as much flow as possible down a s-t path
    - This won't actually work
    - But gives us a sense of what we need to keep track off to improve upon it

Towards a Max-Flow Algorithm

- Greedy strategy:
  - Start with $f(e) = 0$ for each edge
  - Find an $s \rightsquigarrow t$ path P where each edge has $f(e) < c(e)$
  - "Augment" flow (as much as possible) along path P
  - Repeat until you get stuck
- Let's take an example

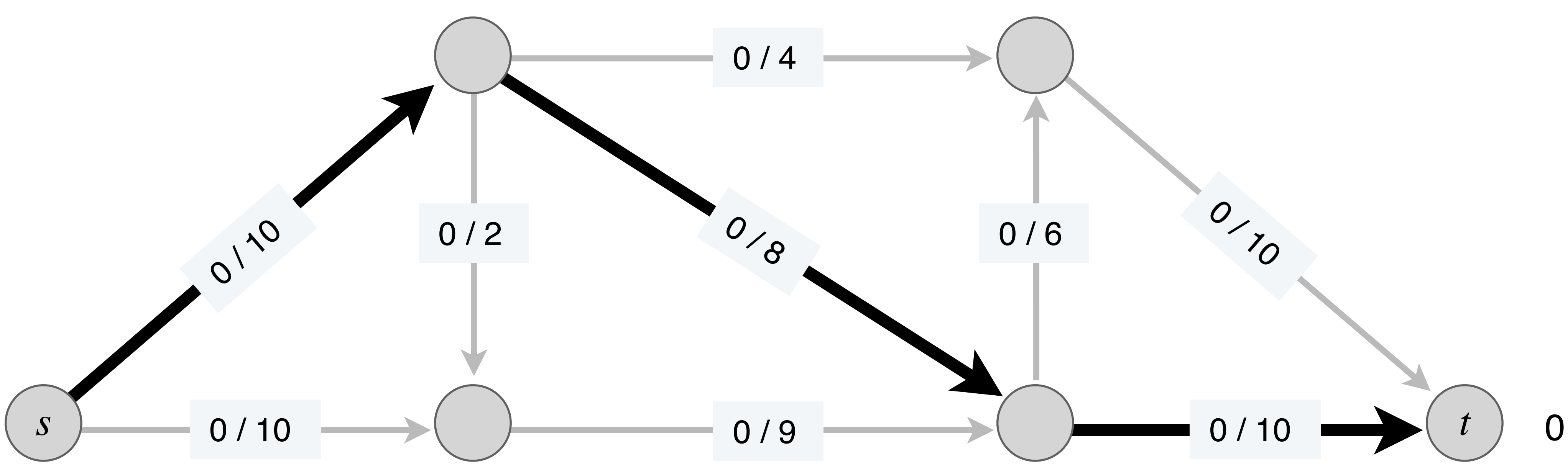
Towards a Max-Flow Algorithm

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



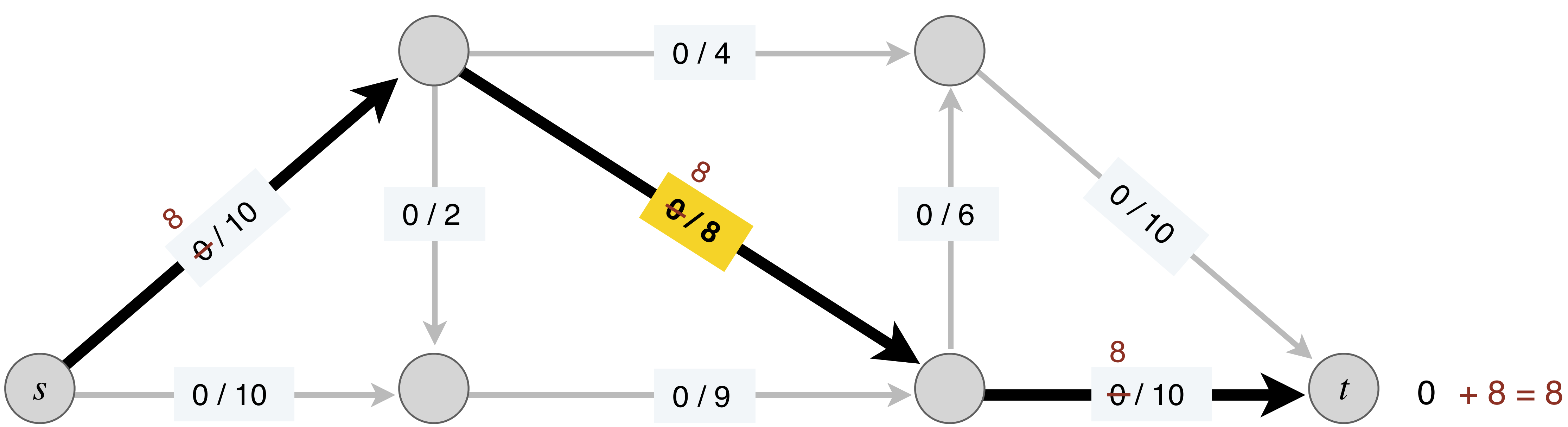
Towards a Max-Flow Algorithm

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



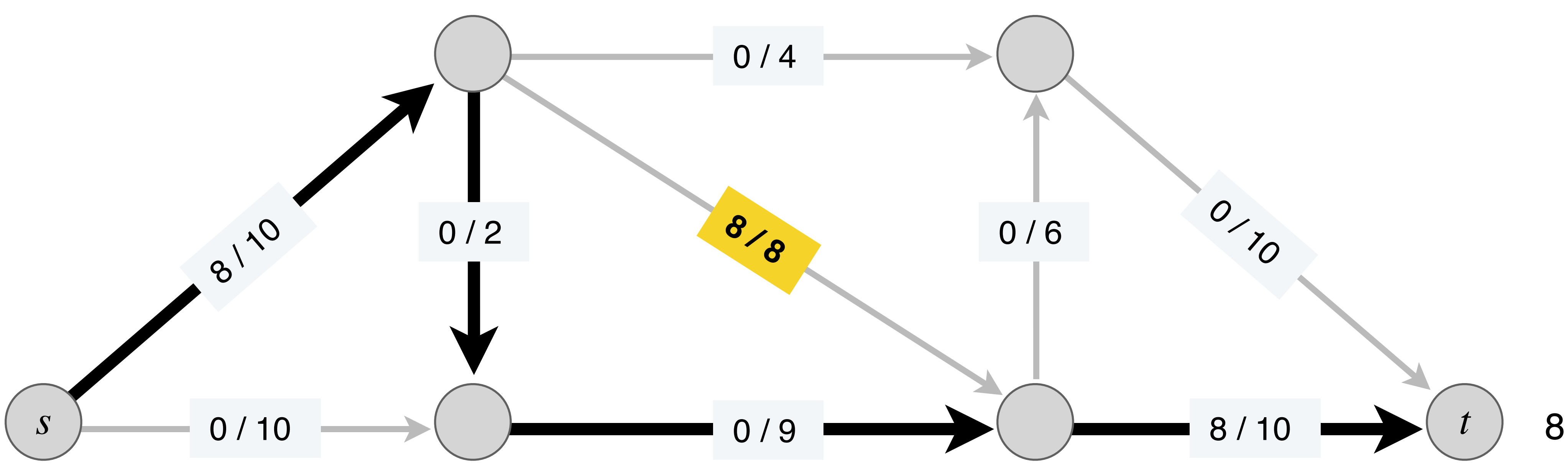
Towards a Max-Flow Algorithm

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



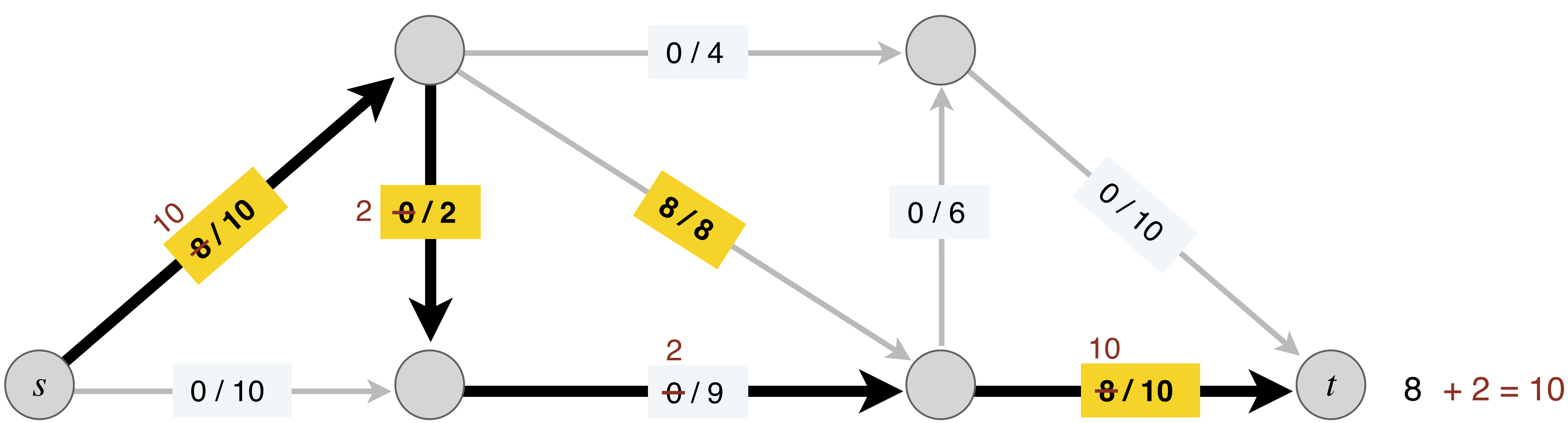
Towards a Max-Flow Algorithm

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



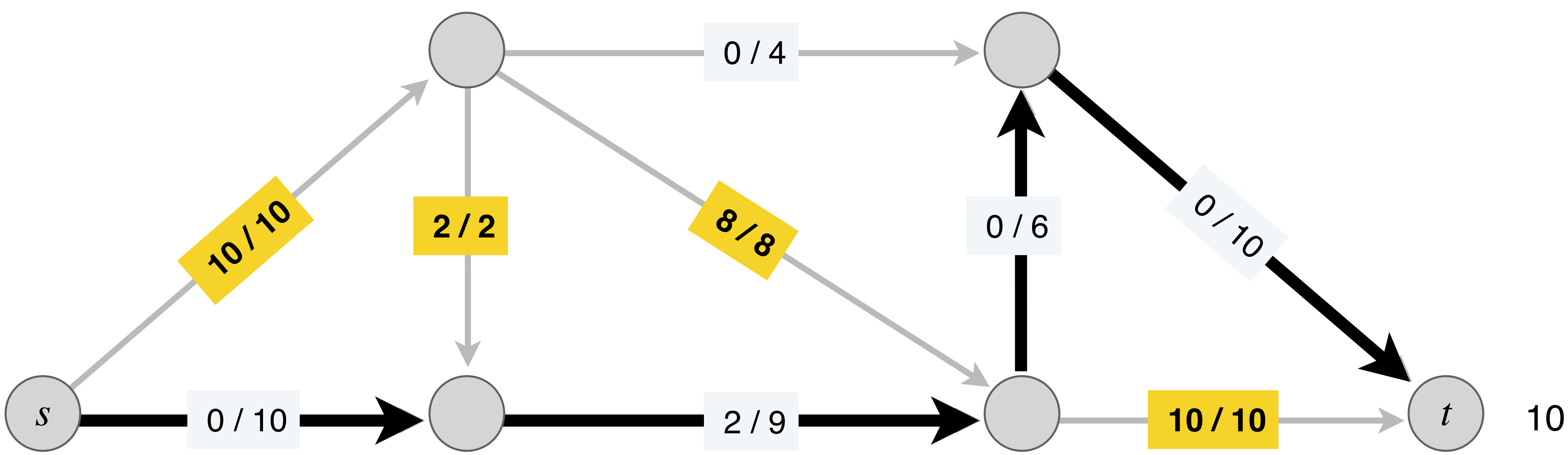
Towards a Max-Flow Algorithm

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



Towards a Max-Flow Algorithm

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

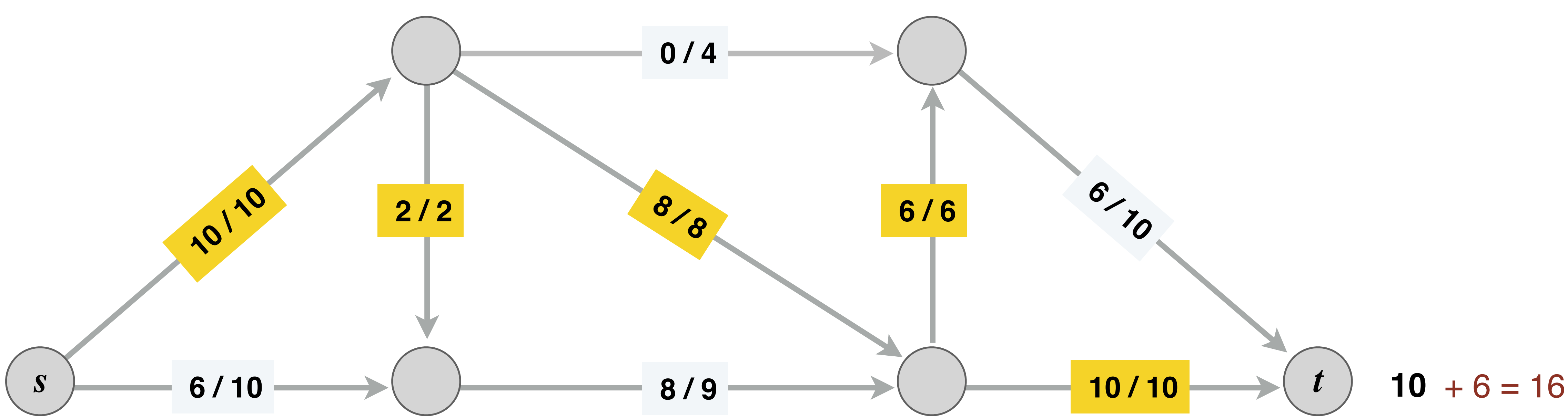


Towards a Max-Flow Algorithm

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

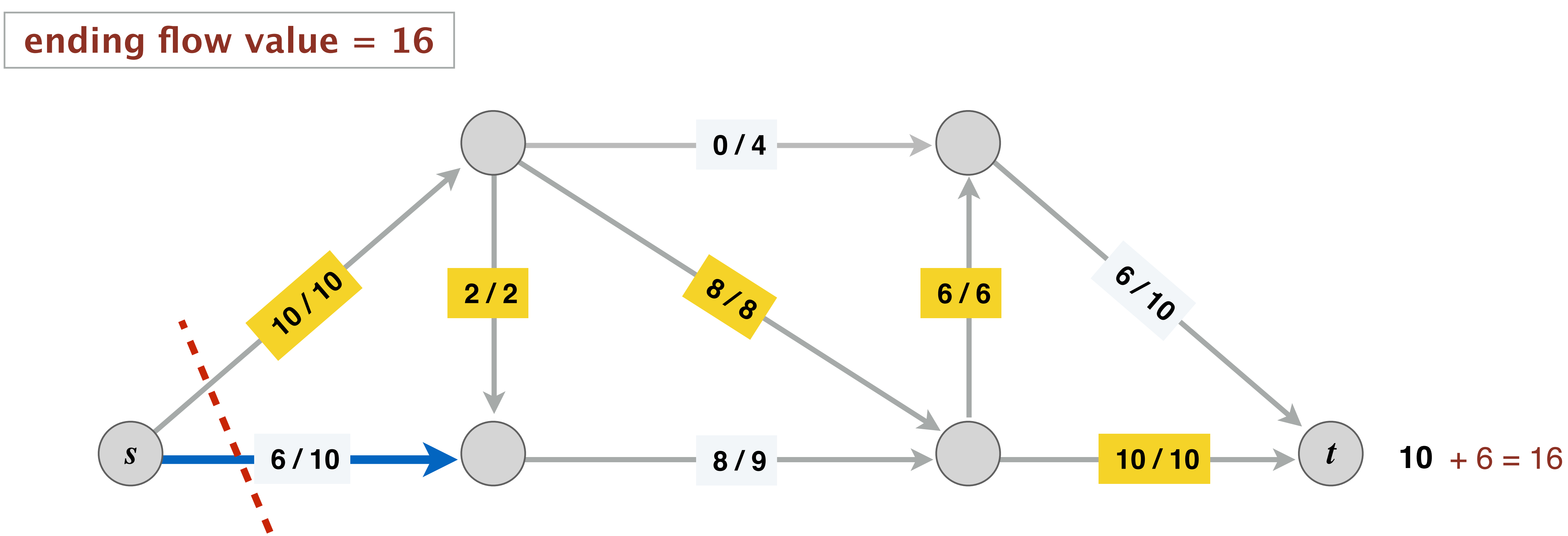
Is this the best we can do?

ending flow value = 16



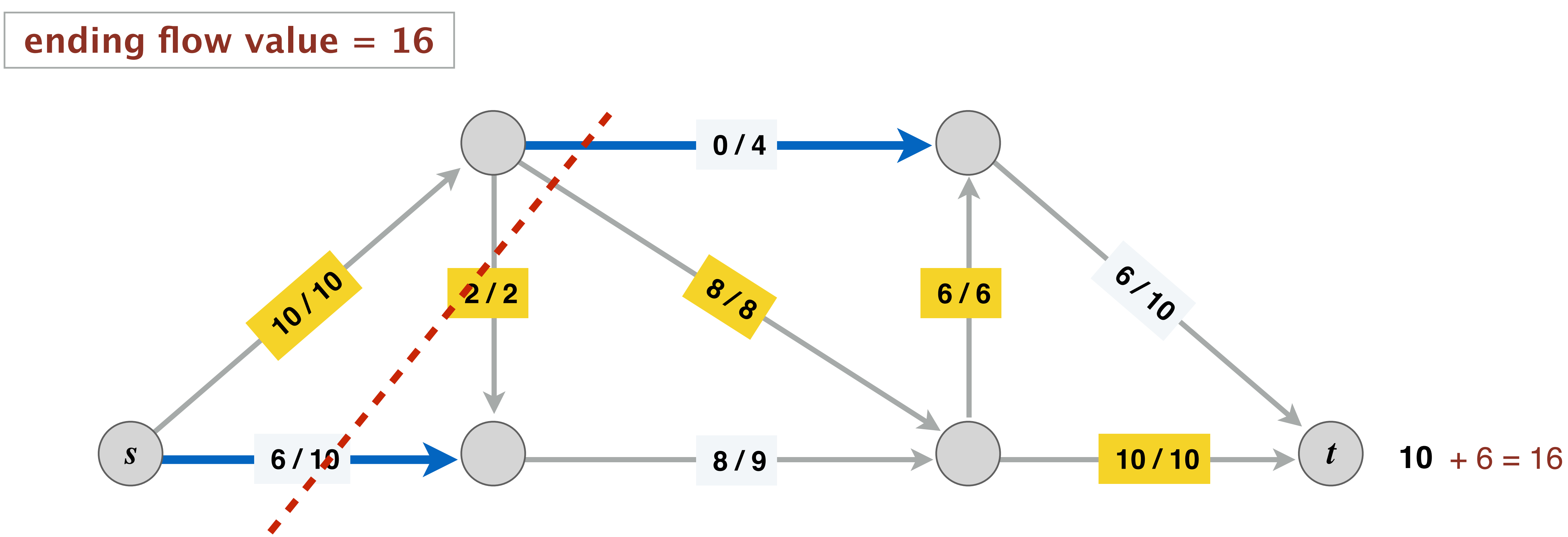
Towards a Max-Flow Algorithm

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



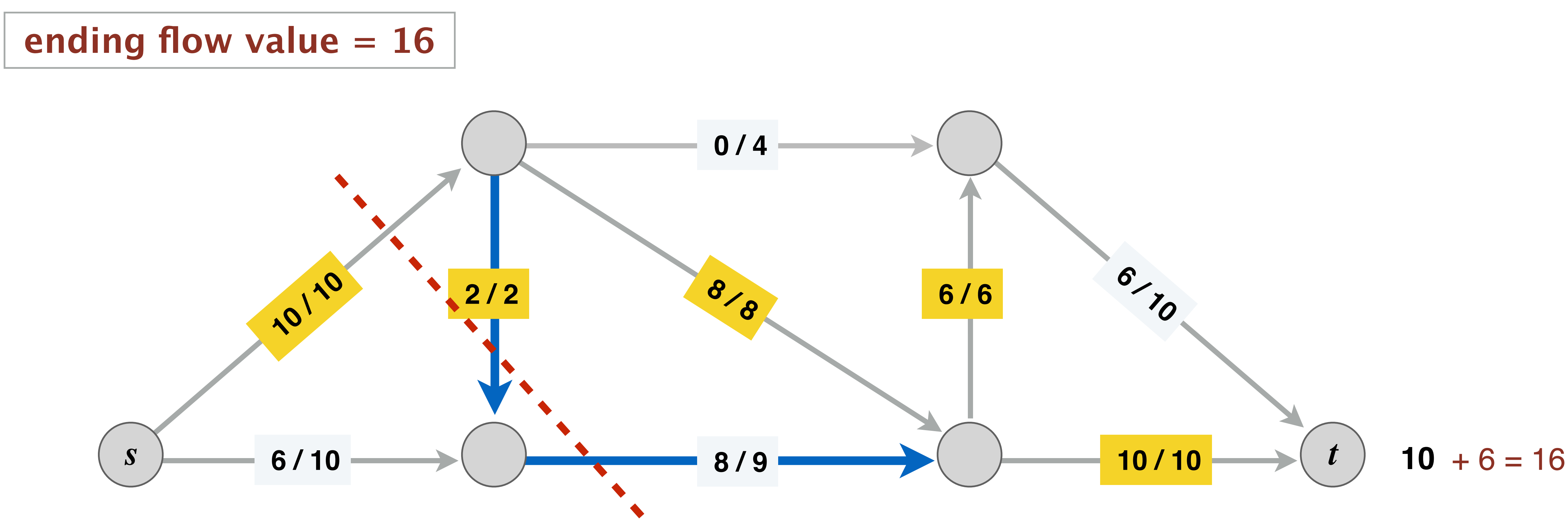
Towards a Max-Flow Algorithm

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



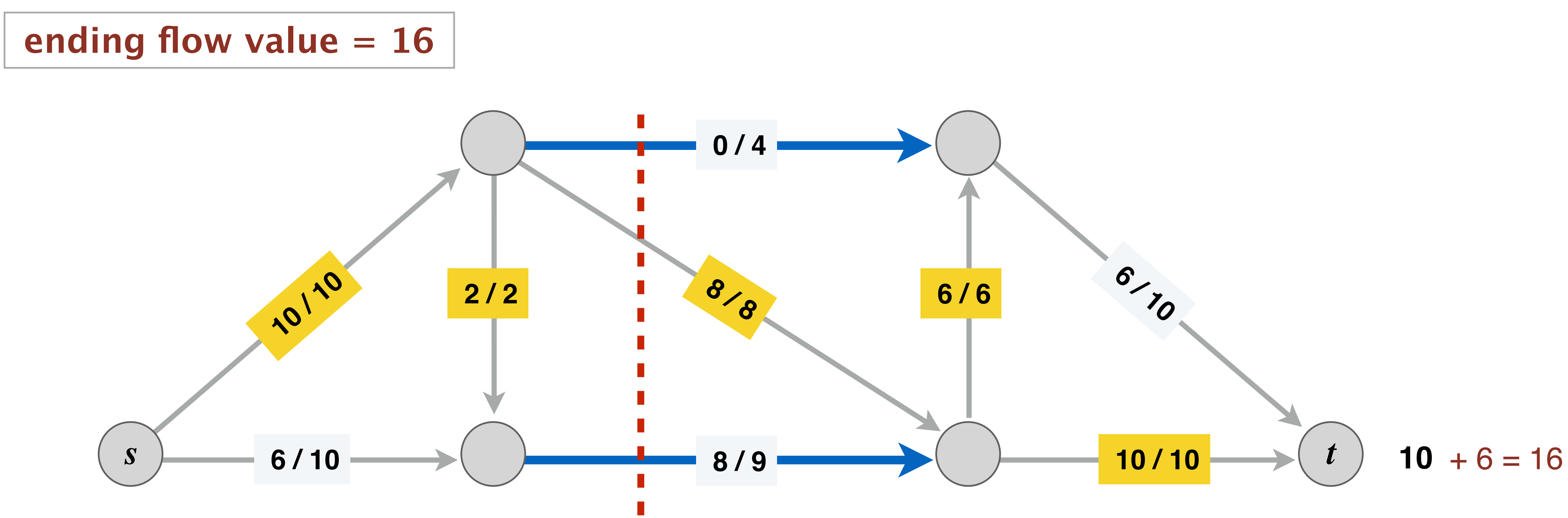
Towards a Max-Flow Algorithm

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



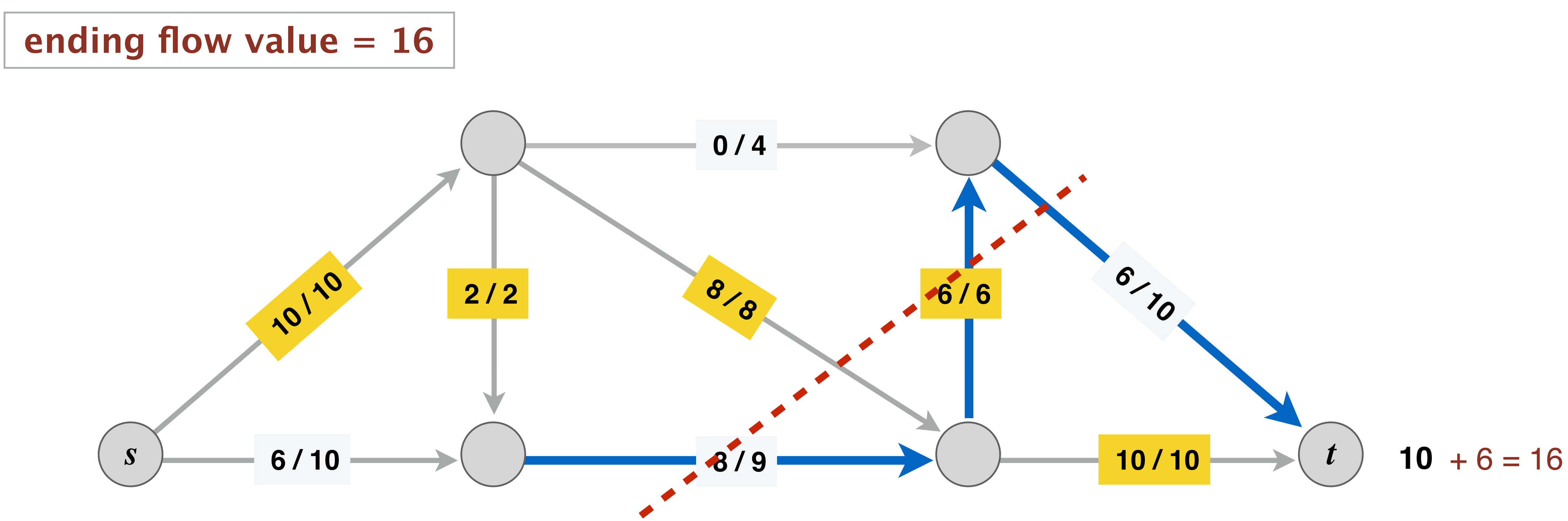
Towards a Max-Flow Algorithm

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



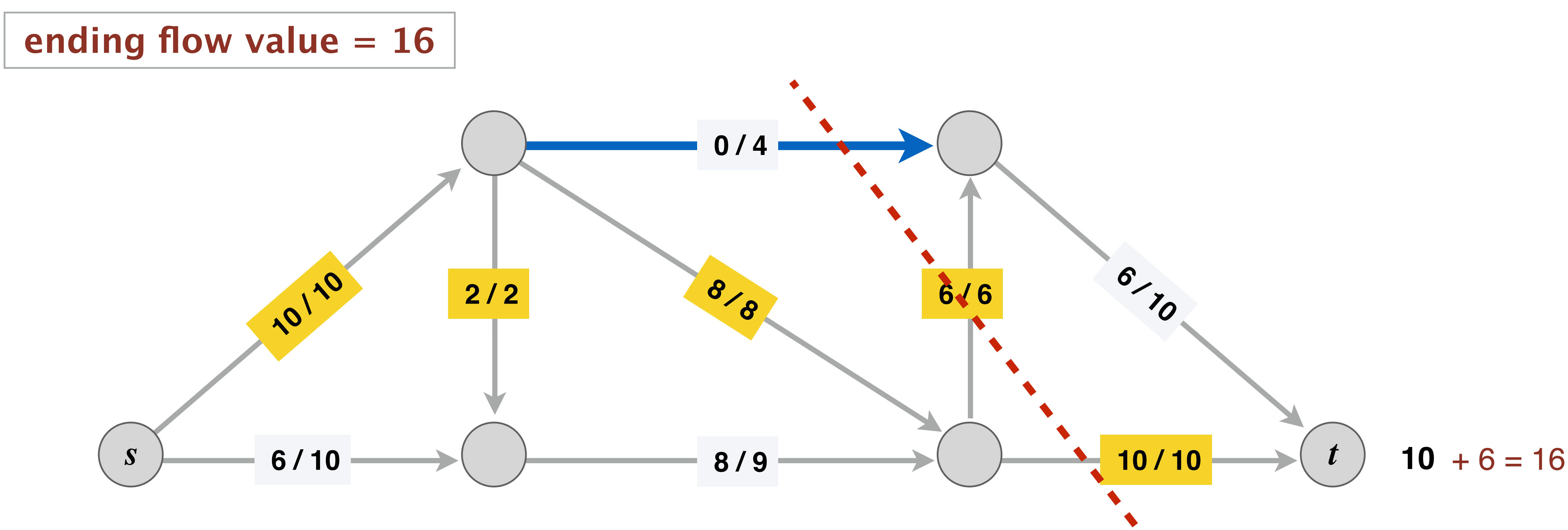
Towards a Max-Flow Algorithm

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



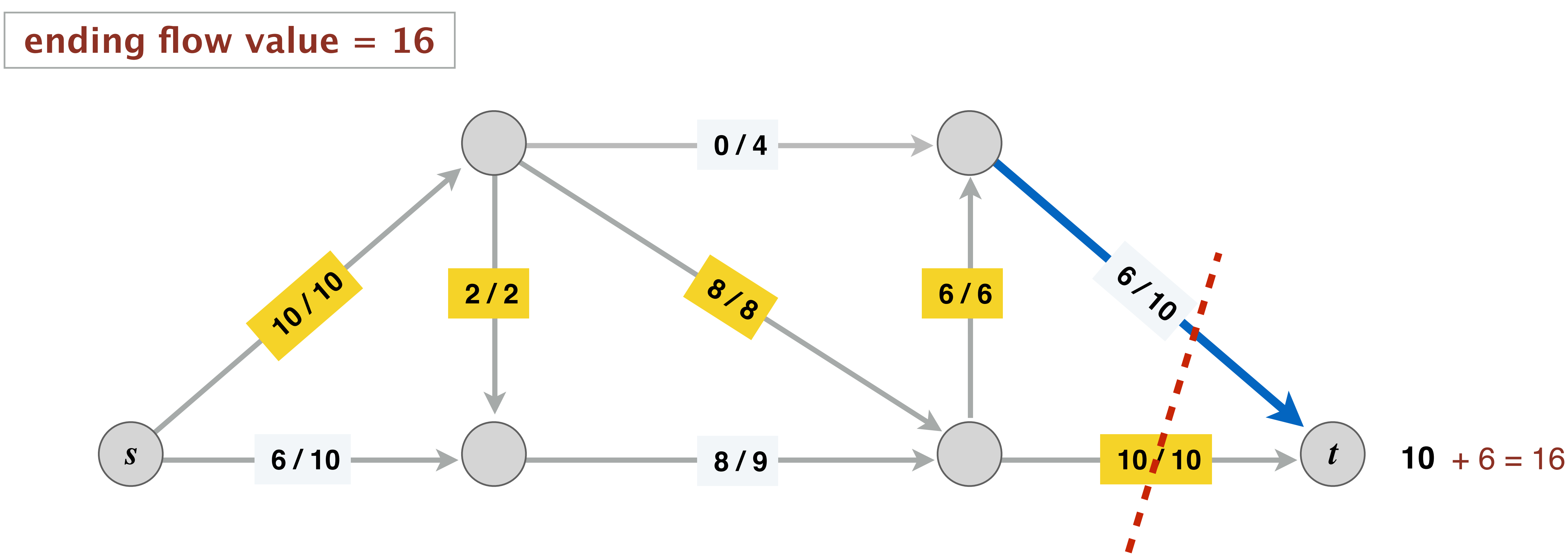
Towards a Max-Flow Algorithm

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



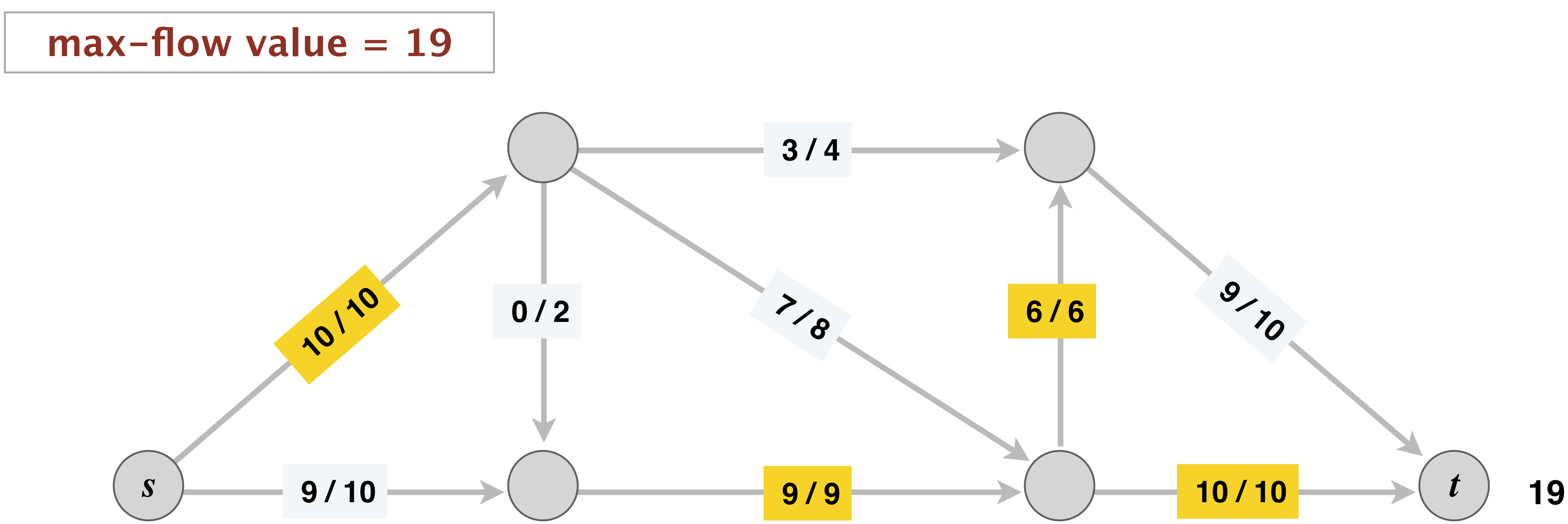
Towards a Max-Flow Algorithm

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



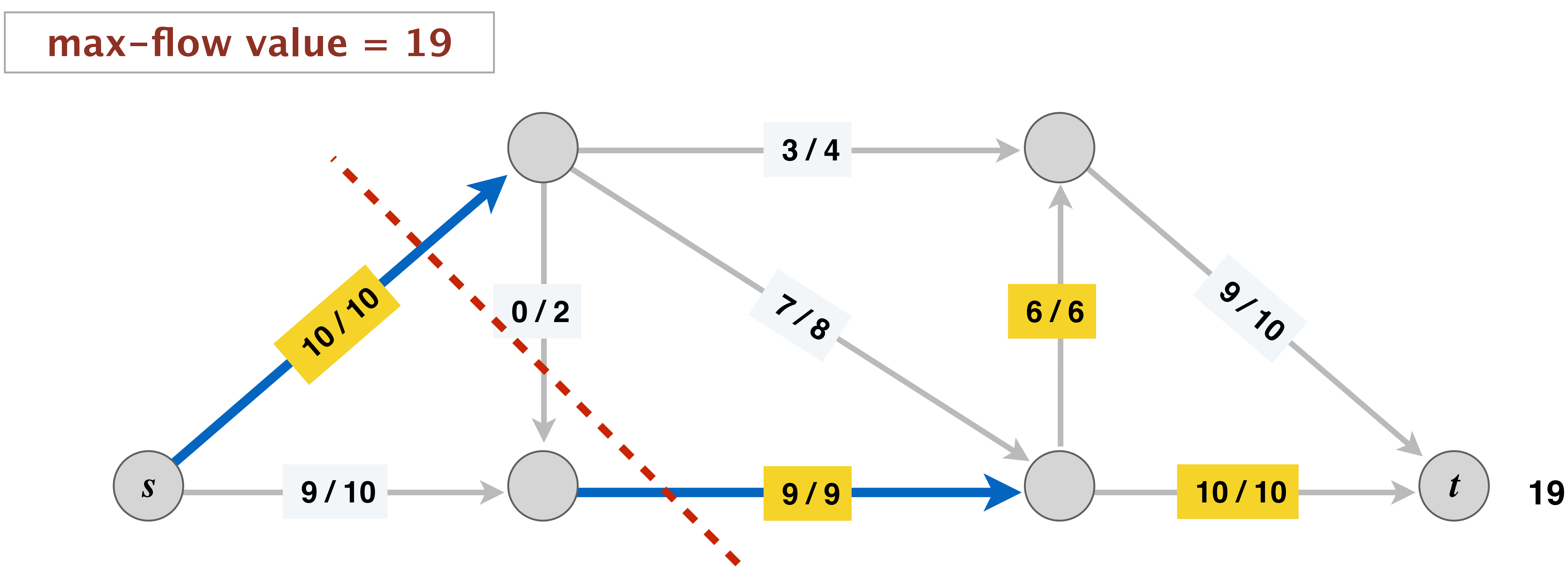
Towards a Max-Flow Algorithm

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



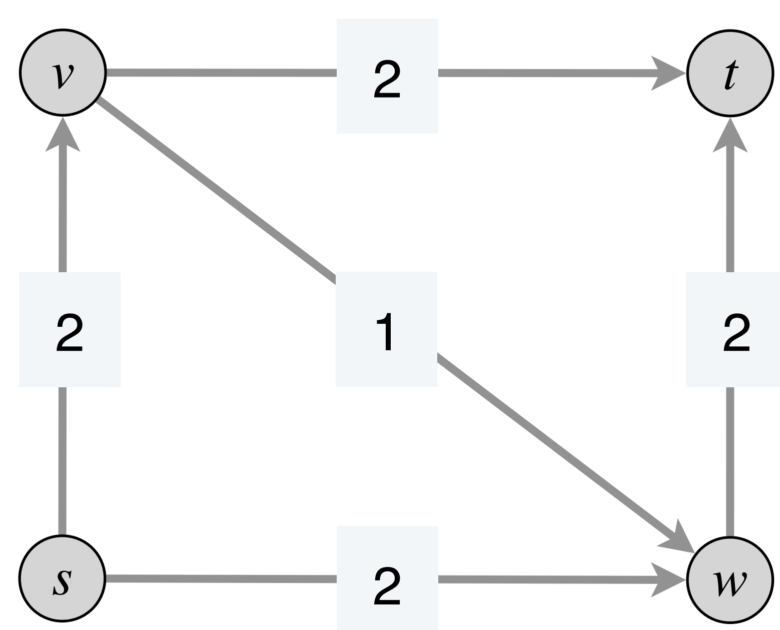
Towards a Max-Flow Algorithm

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



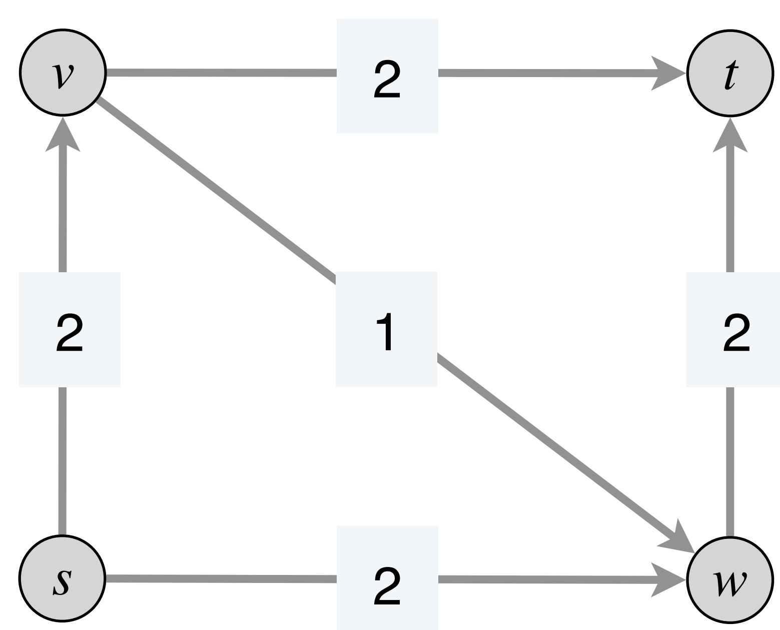
Why Greedy Fails

- **Problem**: greedy can never “undo” a bad flow decision
- Consider the following flow network



Why Greedy Fails

- **Problem**: greedy can never "undo" a bad flow decision
- Consider the following flow network
  - Unique max flow has $f(v \to w) = 0$
  - Greedy could choose $s \to v \to w \to t$ as first P



- **Takeaway**: Need a mechanism to "undo" bad flow decisions

Ford-Fulkerson Algorithm

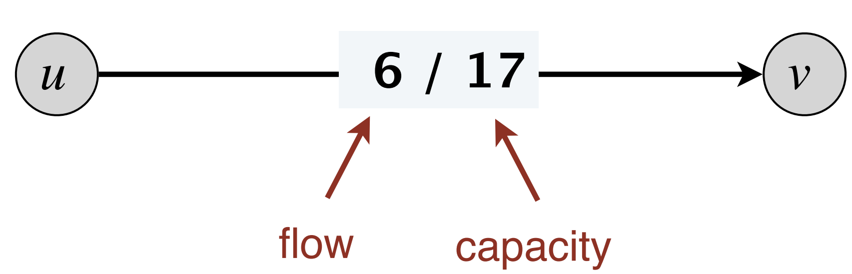
Ford Fulkerson: Idea

- Want to make “forward progress” while letting ourselves undo previous decisions if they’re getting in our way
- **Idea**: keep track of where we can push flow
  - Can push more flow along an edge with remaining capacity
  - Can also push flow “back” along an edge that already has flow down it
- Need a way to systematically track these decisions

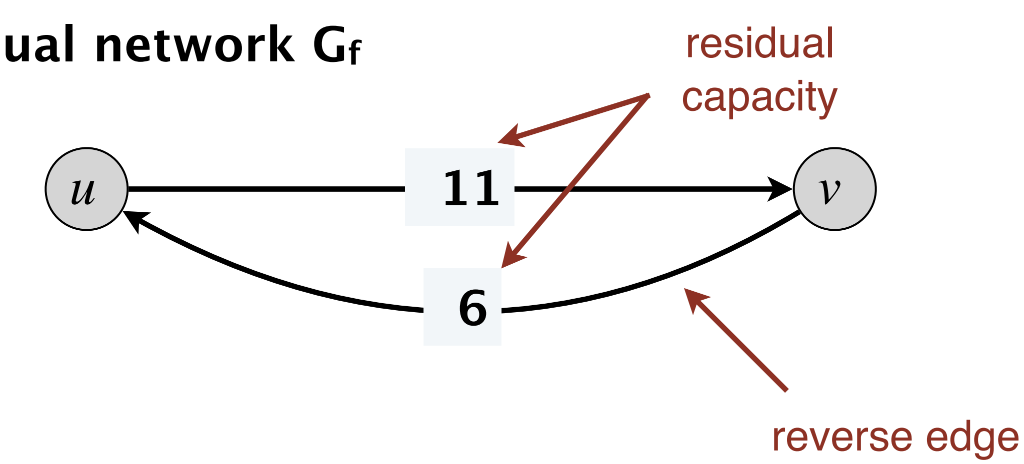
Residual Graph

- Given flow network $G = (V, E, c)$ and a feasible flow f on G, **the residual graph** $G_f = (V, E_f, c_f)$ is defined as:
    - Vertices in G_f same as G
    - **(Forward edge)** For $e \in E$ with residual capacity $c(e) - f(e) > 0$, create $e \in E_f$ with capacity $c(e) - f(e)$
    - **(Backward edge)** For $e \in E$ with $f(e) > 0$, create $e_{\text{reverse}} \in E_f$ with capacity $f(e)$

original flow network G



residual network Gf



Flow Algorithm Idea

- Now we have a residual graph that lets us make forward progress or push back existing flow
- We will look for $s \rightsquigarrow t$ paths in G_f rather than G
- Once we have a path, we will "augment" flow along it similar to greedy
    - find bottleneck capacity edge on the path and push that much flow through it in G_f
- When we translate this back to G, this means:
    - We increment existing flow on a forward edge
    - Or we decrement flow on a backward edge

Augmenting Path & Flow

- An **augmenting path** P is a **simple** $s \rightsquigarrow t$ path in the residual graph G_f
- The **bottleneck capacity** b of an augmenting path P is the minimum capacity of any edge in P.

The path P is in G_f

AUGMENT(f, P)

$b \leftarrow$ bottleneck capacity of augmenting path P.

FOREACH edge $e \in P$:

  IF ($e \in E$, *that is, e is forward edge*)

    Increase *f(e)* in G by b

  ELSE

    Decrease *f(e)* in G by b

RETURN f.

Updating flow in G

Ford-Fulkerson Algorithm

- Start with $f(e) = 0$ for each edge $e \in E$
- Find a simple $s \rightsquigarrow t$ path P in the residual network G_f
- Augment flow along path P by bottleneck capacity b
- Repeat until you get stuck

FORD–FULKERSON(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an s↝t path P in G_f)

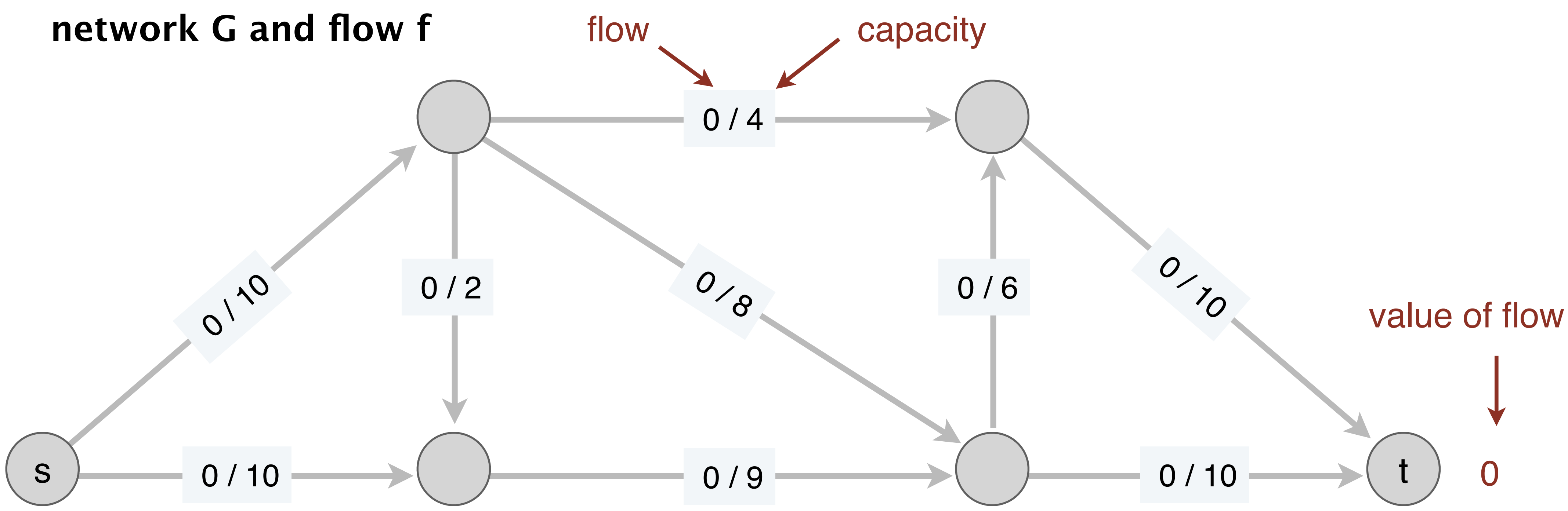
  $f \leftarrow$ AUGMENT(f, P).

  Update G_f.

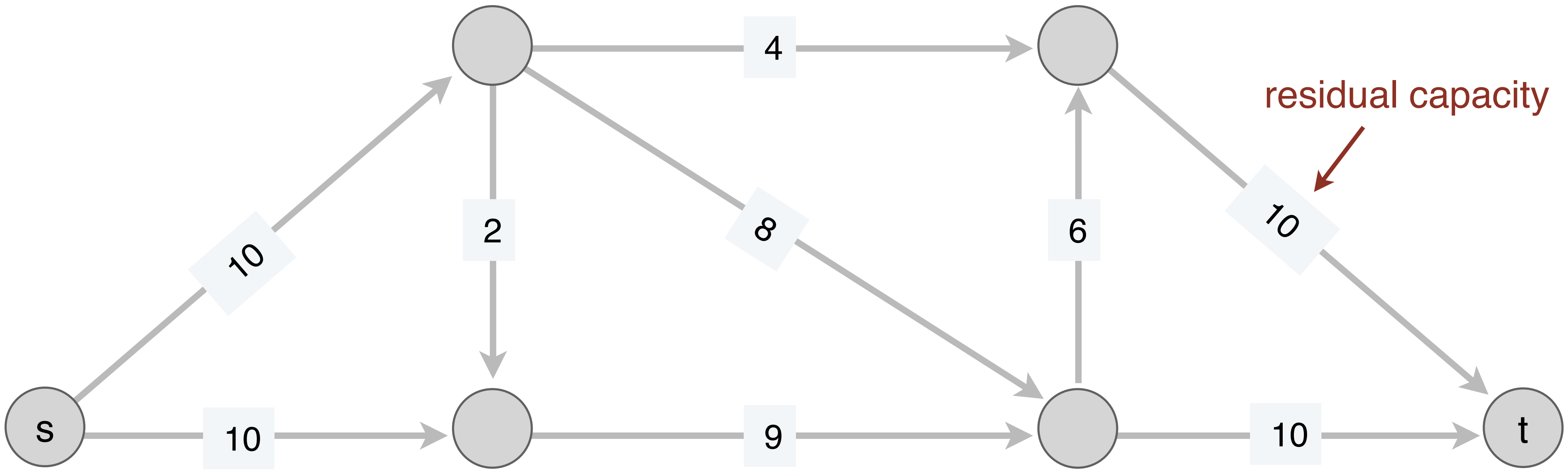
RETURN f.

Ford-Fulkerson Example

network G and flow f

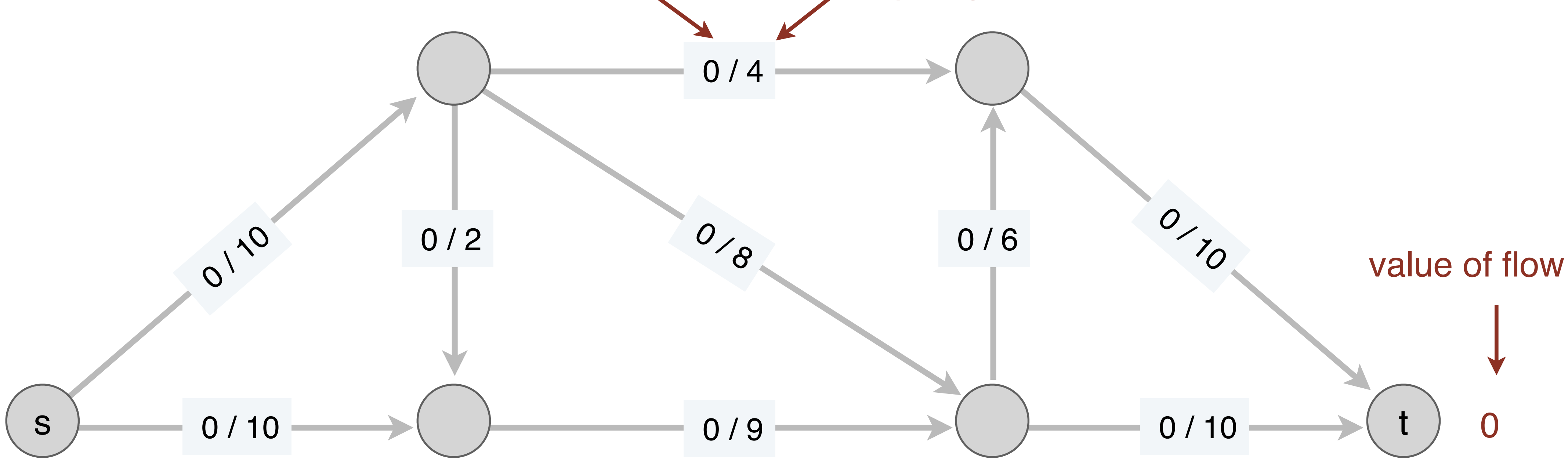


residual network G_f

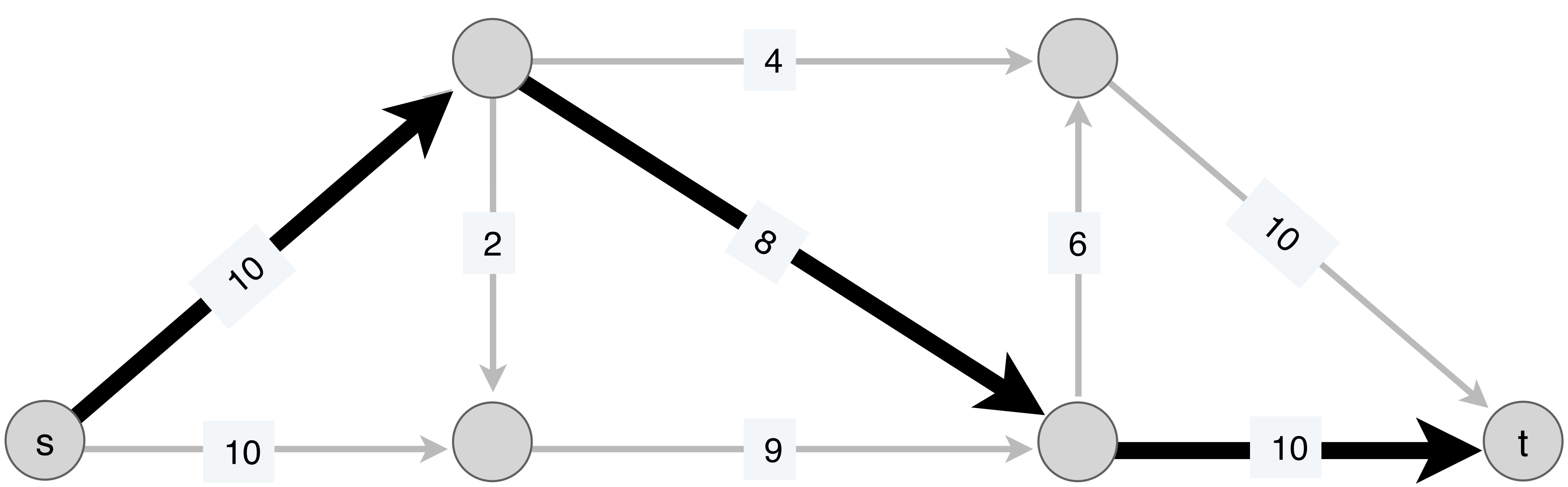


Ford-Fulkerson Example

network G and flow f

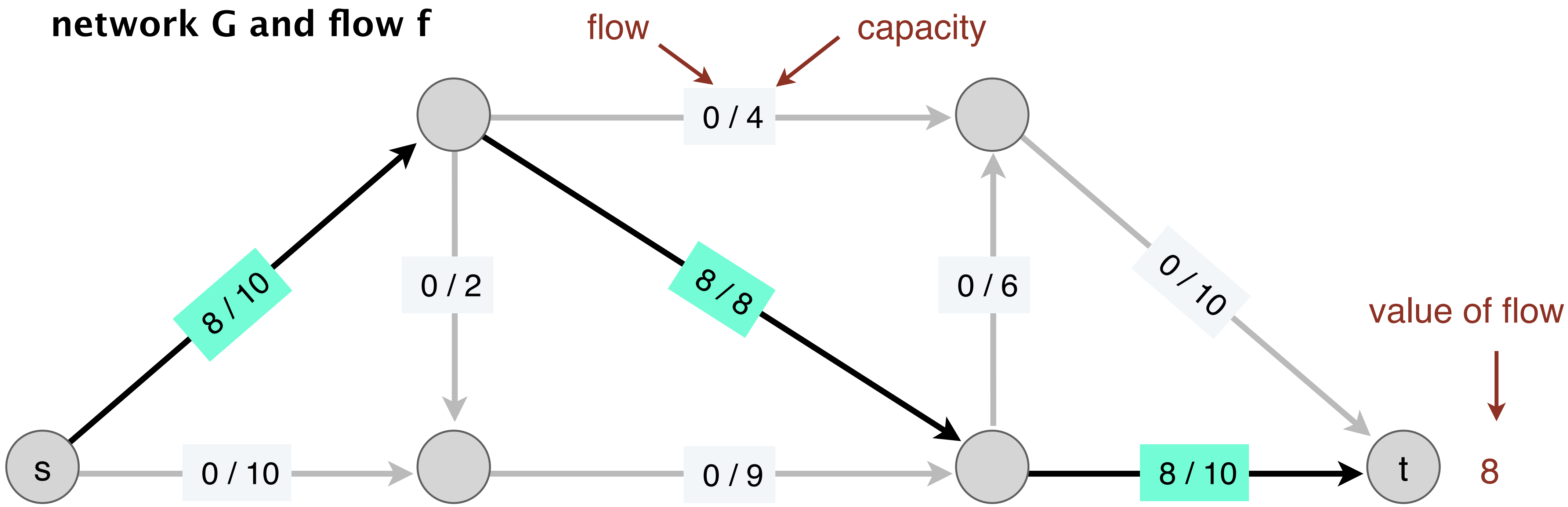


P in residual network G_f

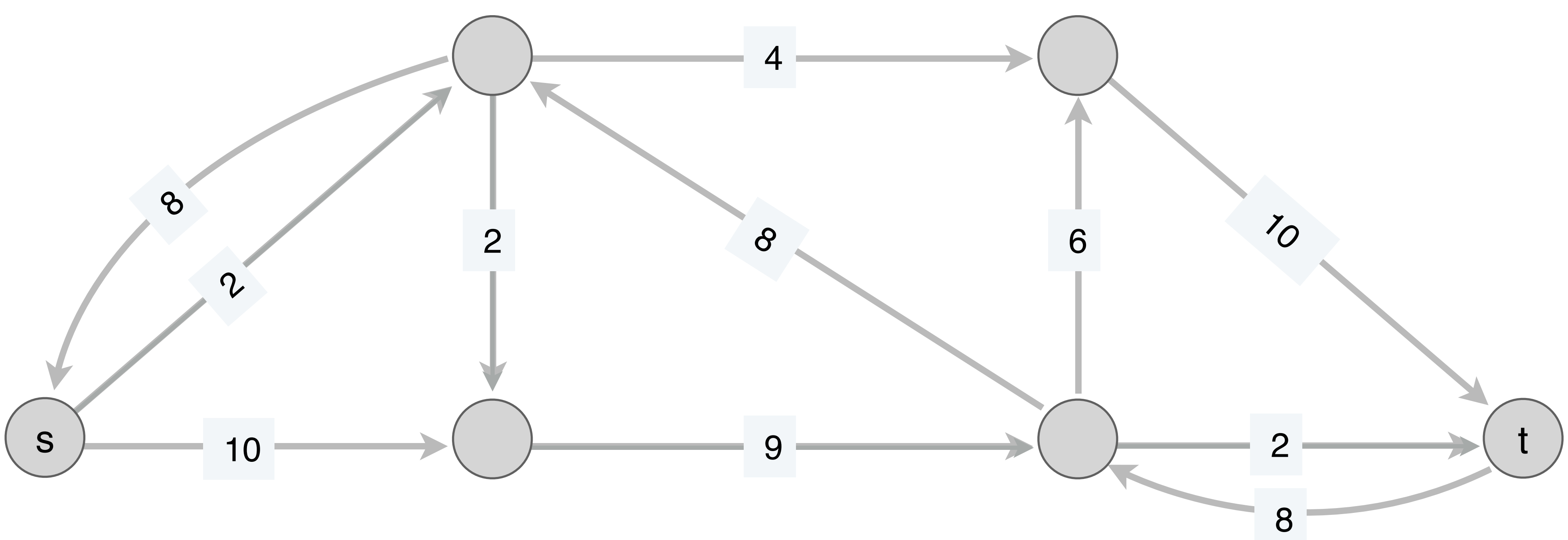


Ford-Fulkerson Example

network G and flow f

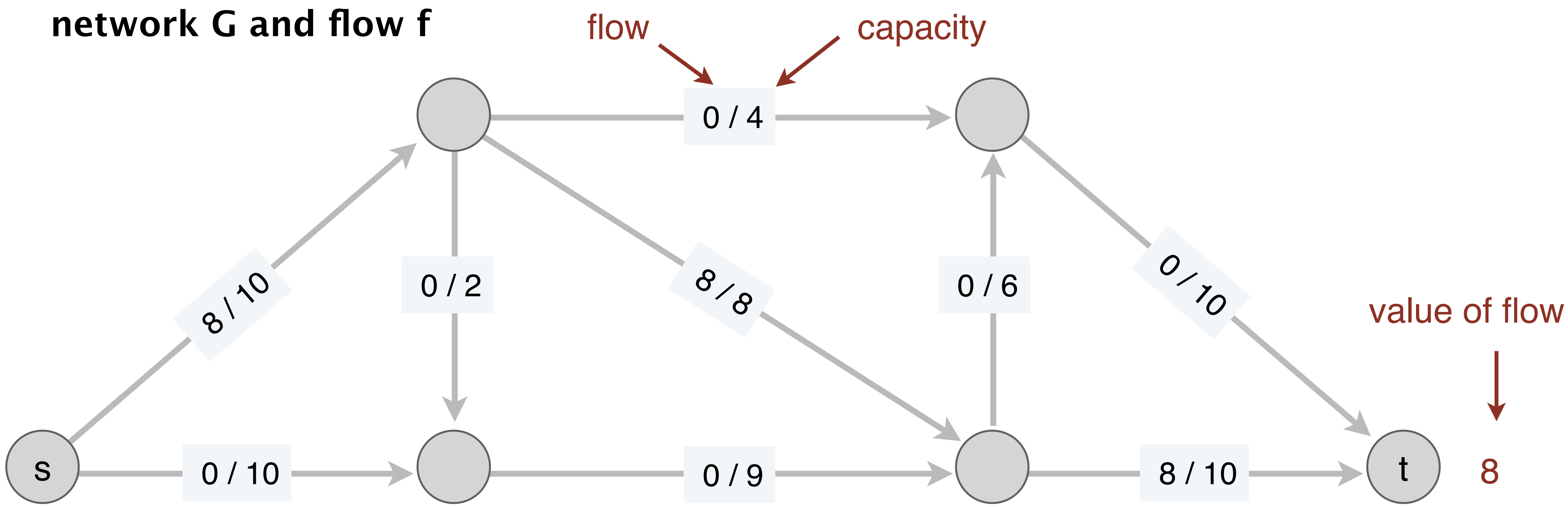


residual network G_f

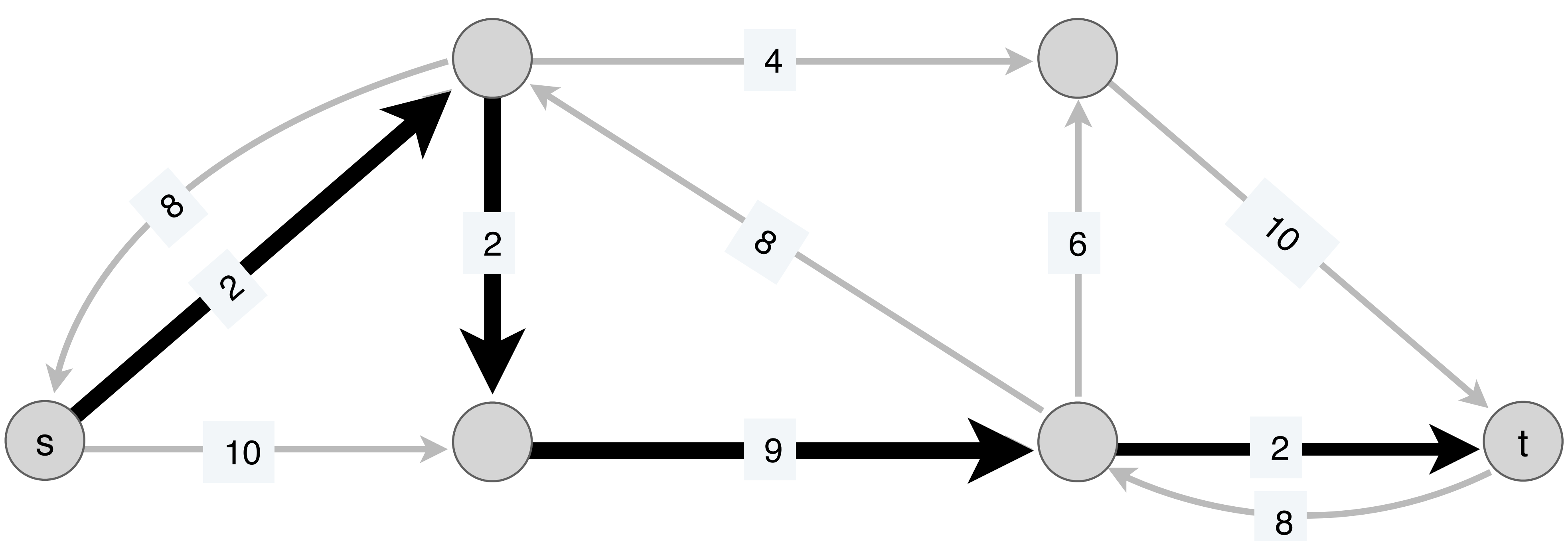


Ford-Fulkerson Example

network G and flow f

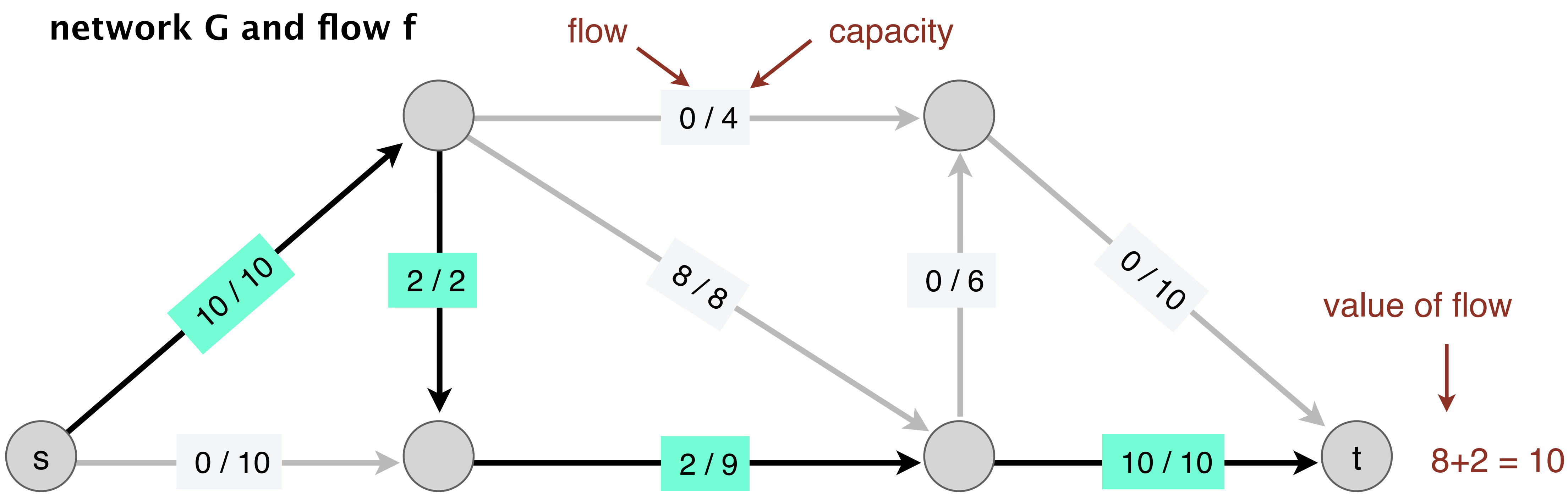


P in residual network G_f

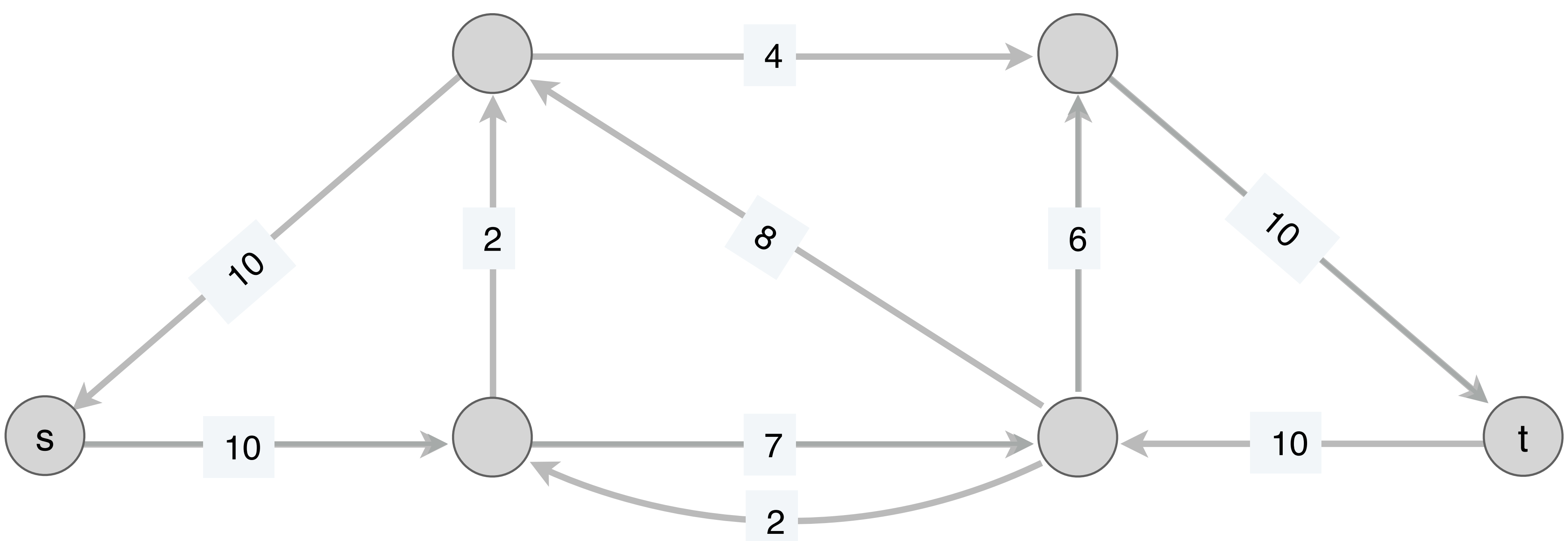


Ford-Fulkerson Example

network G and flow f

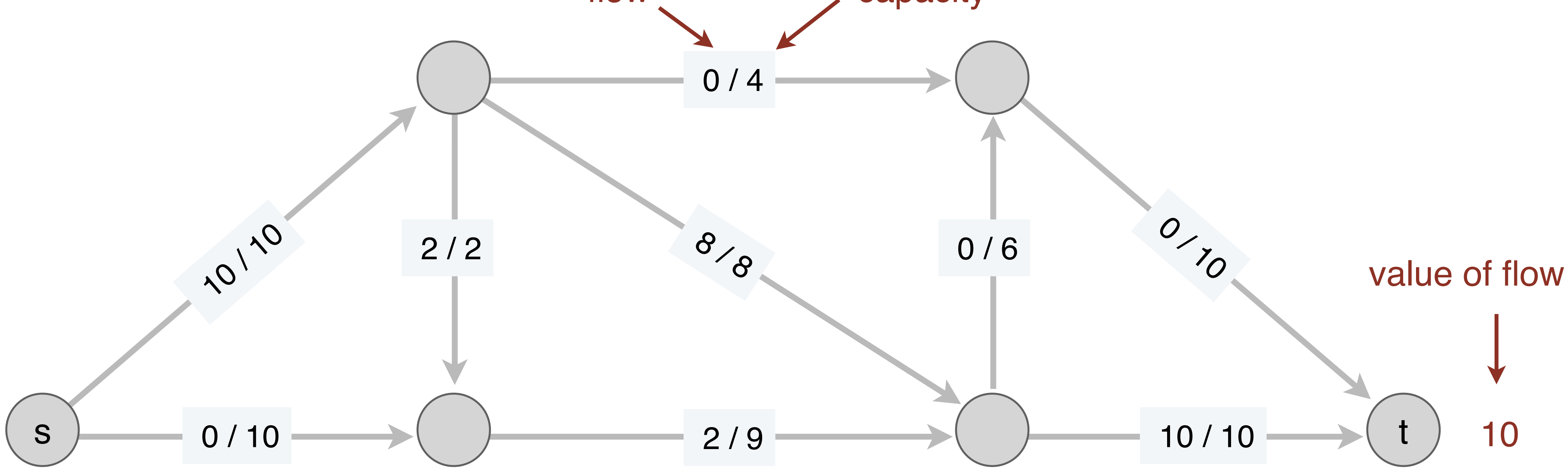


residual network G_f

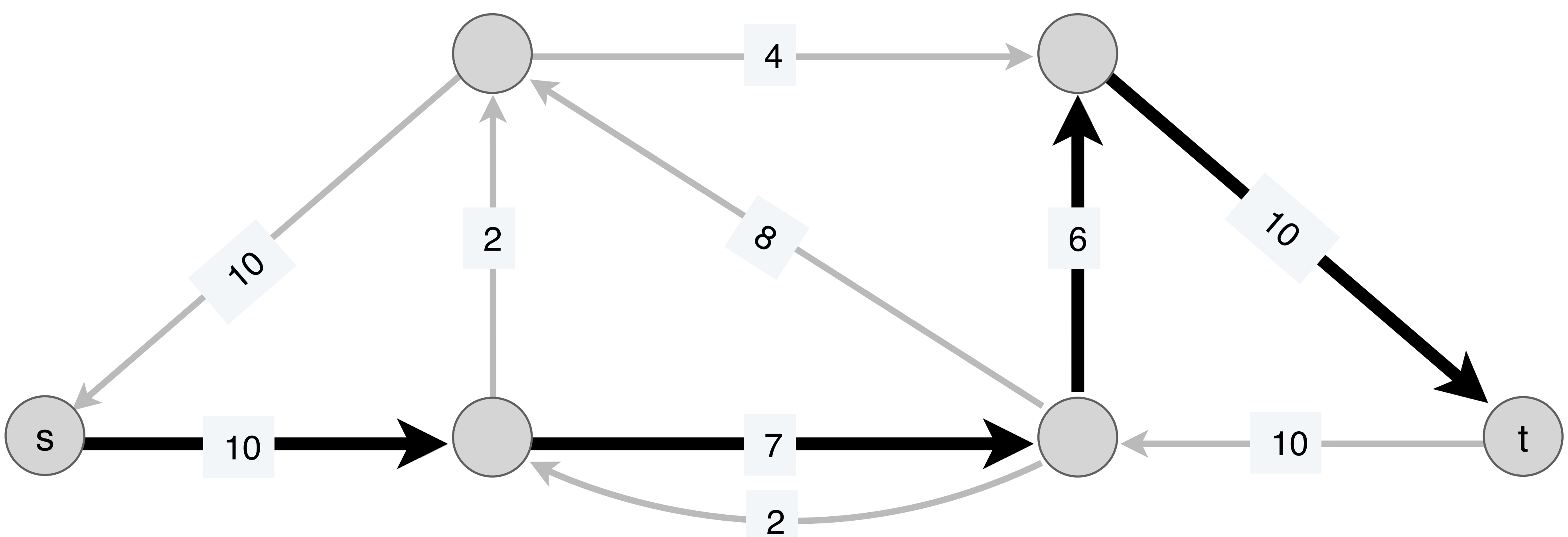


Ford-Fulkerson Example

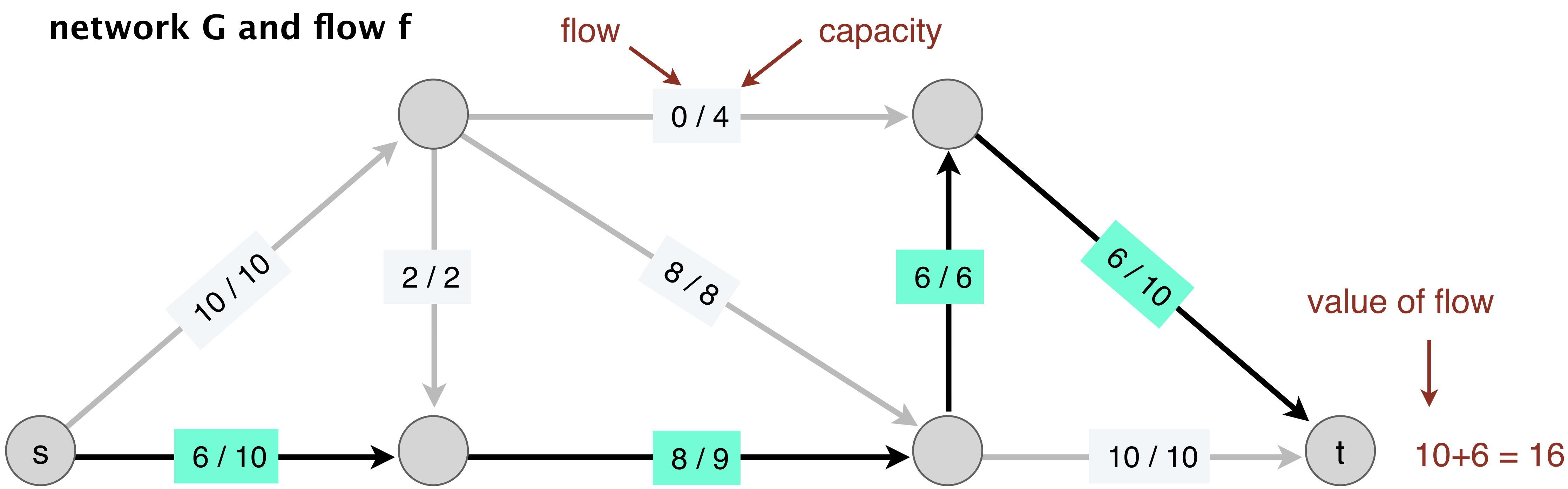
network G and flow f

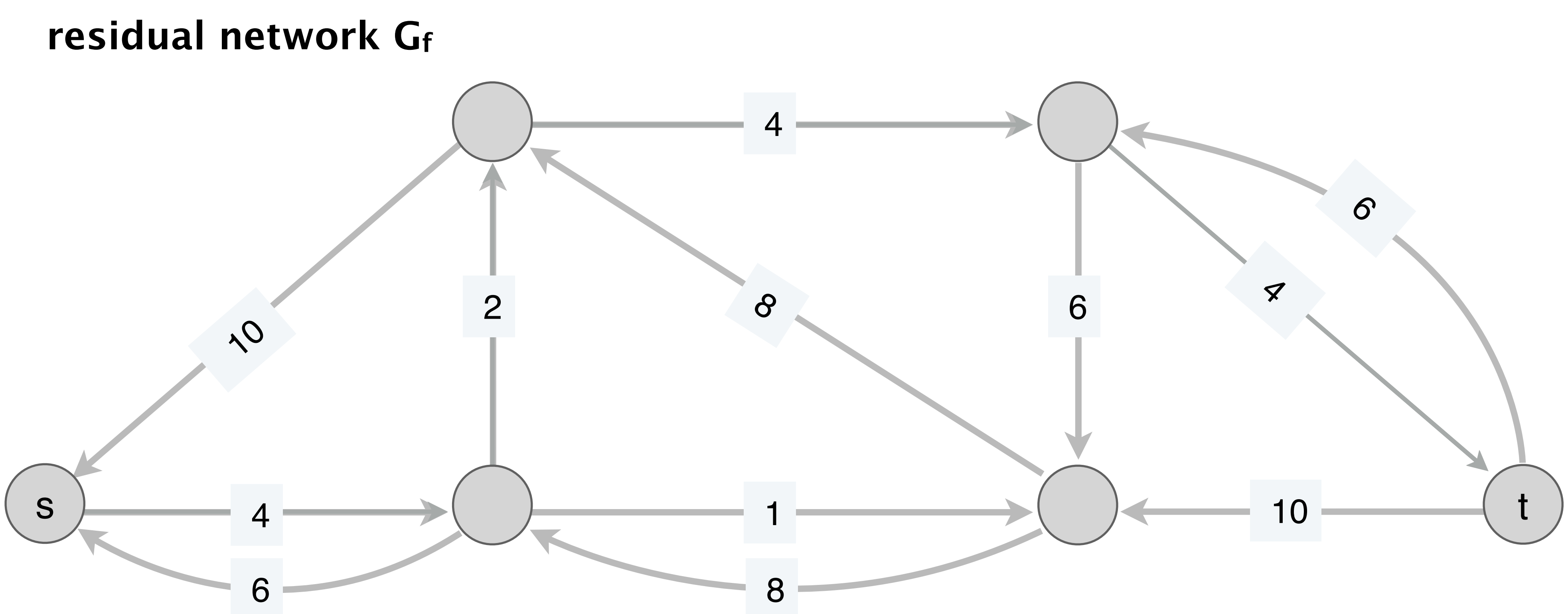


P in residual network G_f

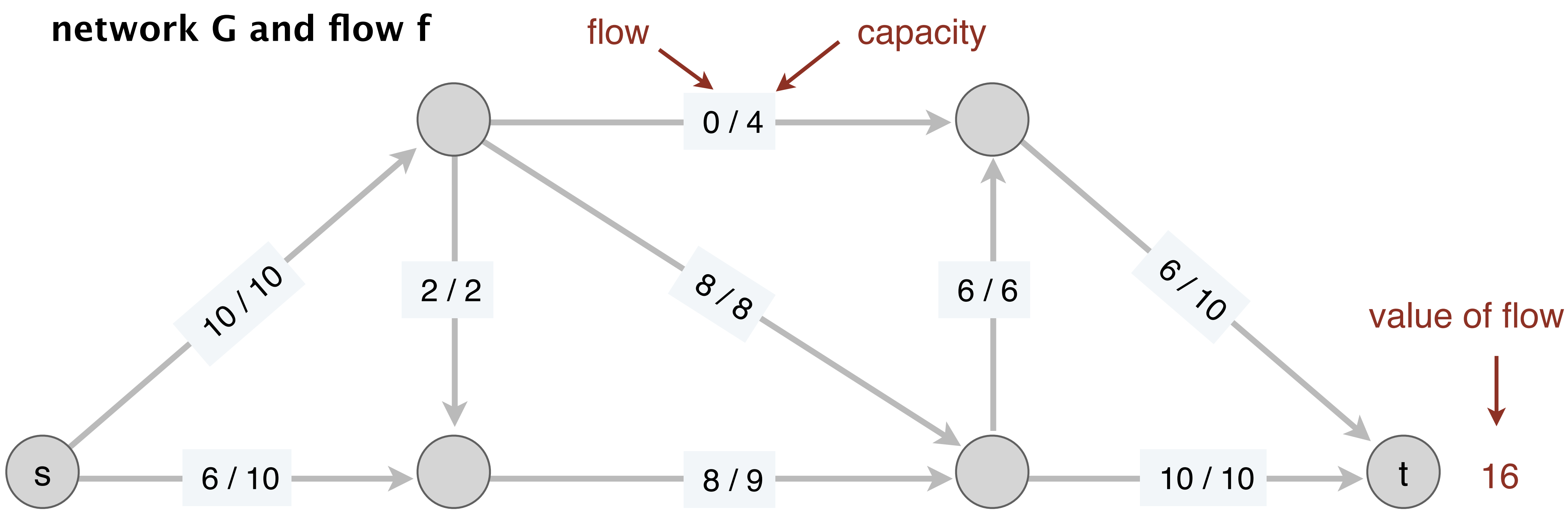


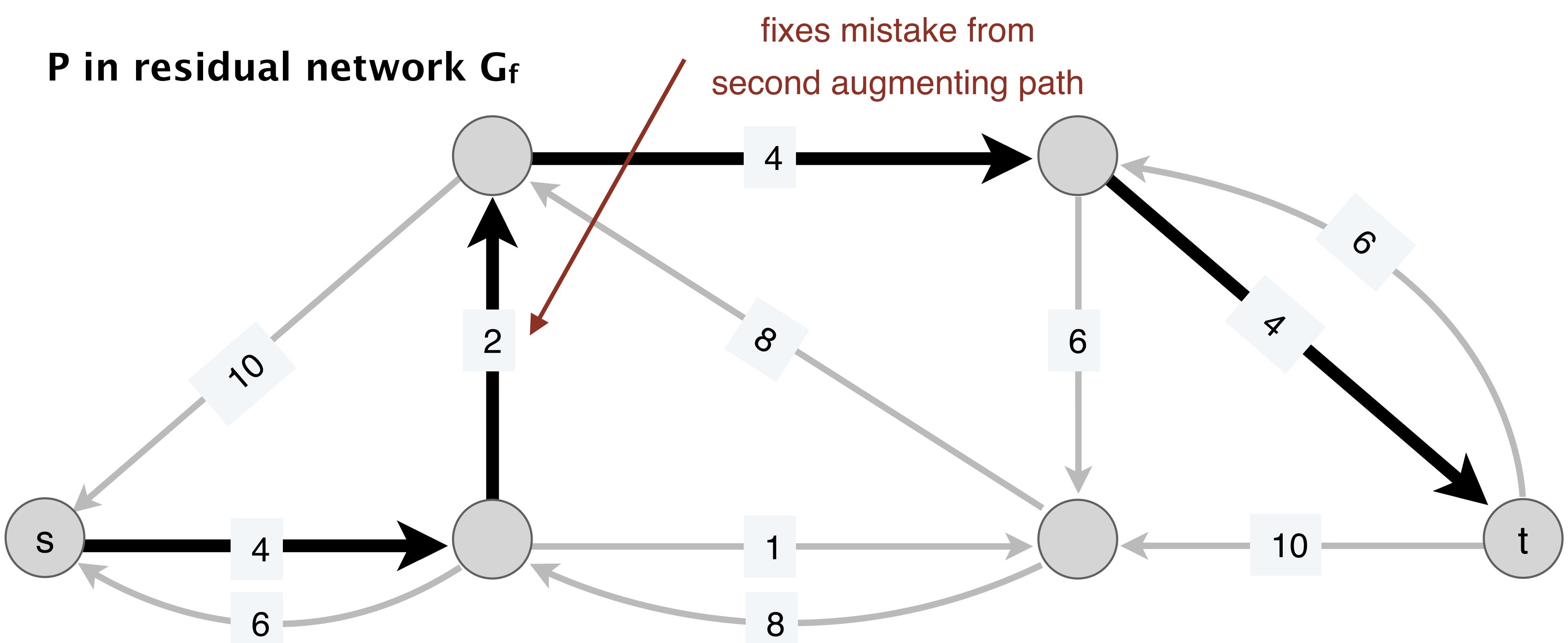
Ford-Fulkerson Example



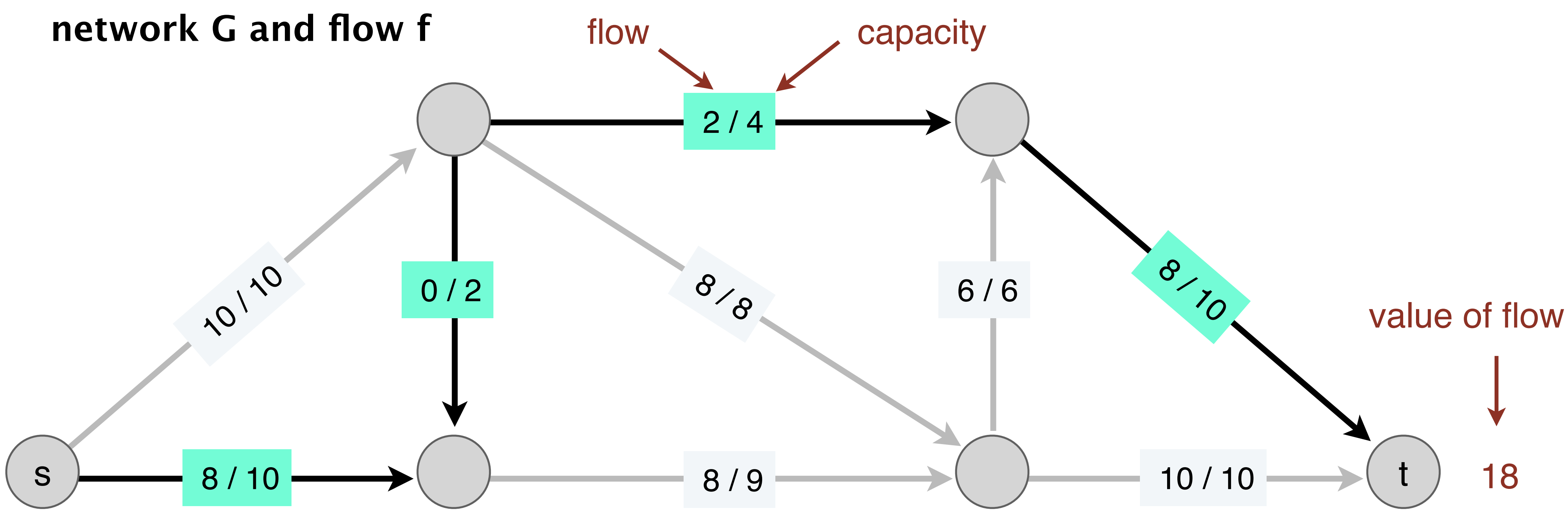


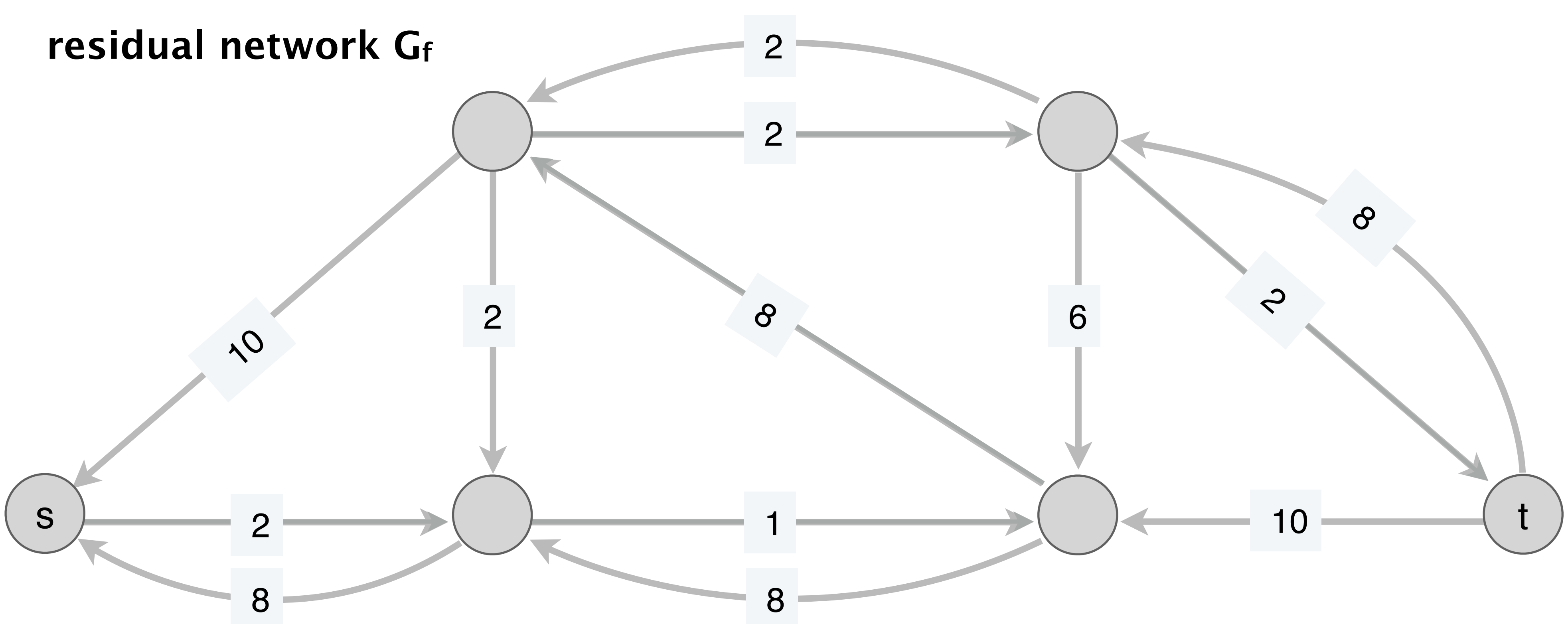
Ford-Fulkerson Example





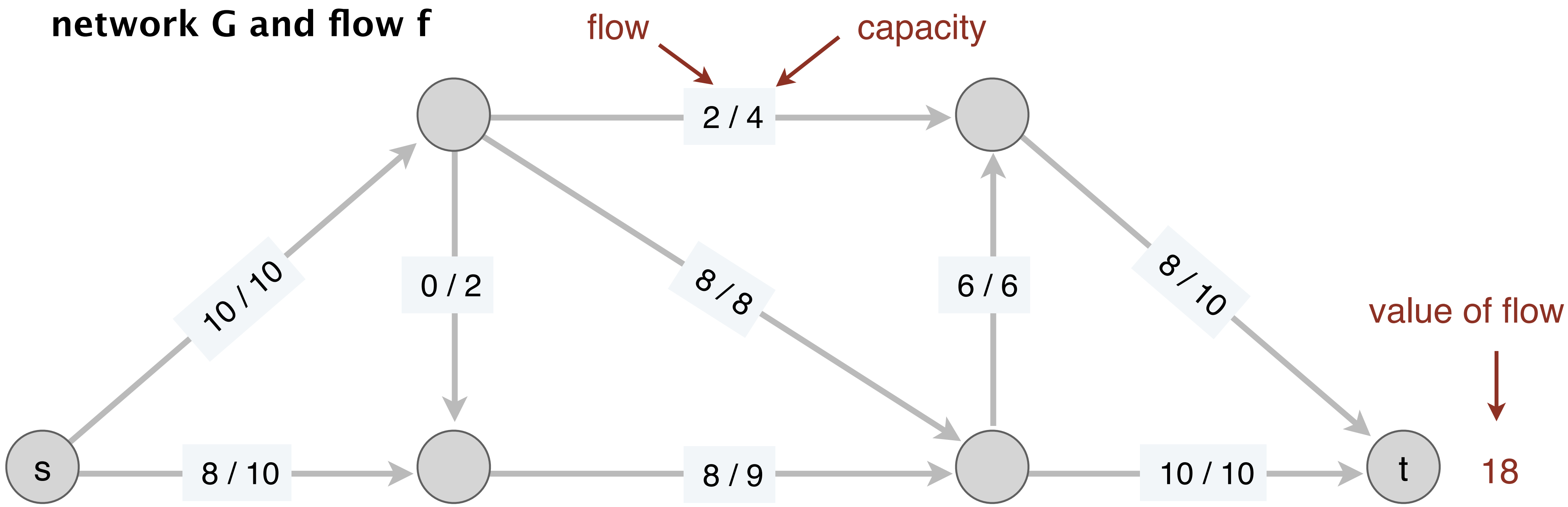
Ford-Fulkerson Example



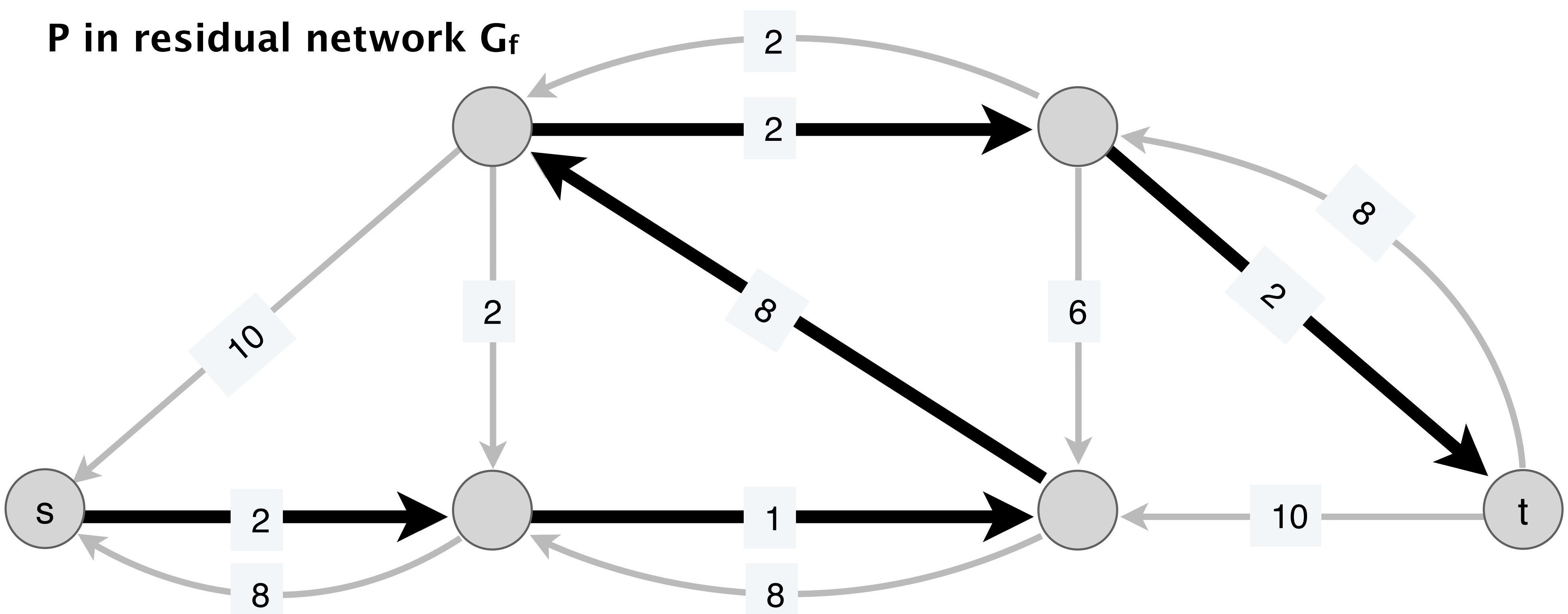


Ford-Fulkerson Example

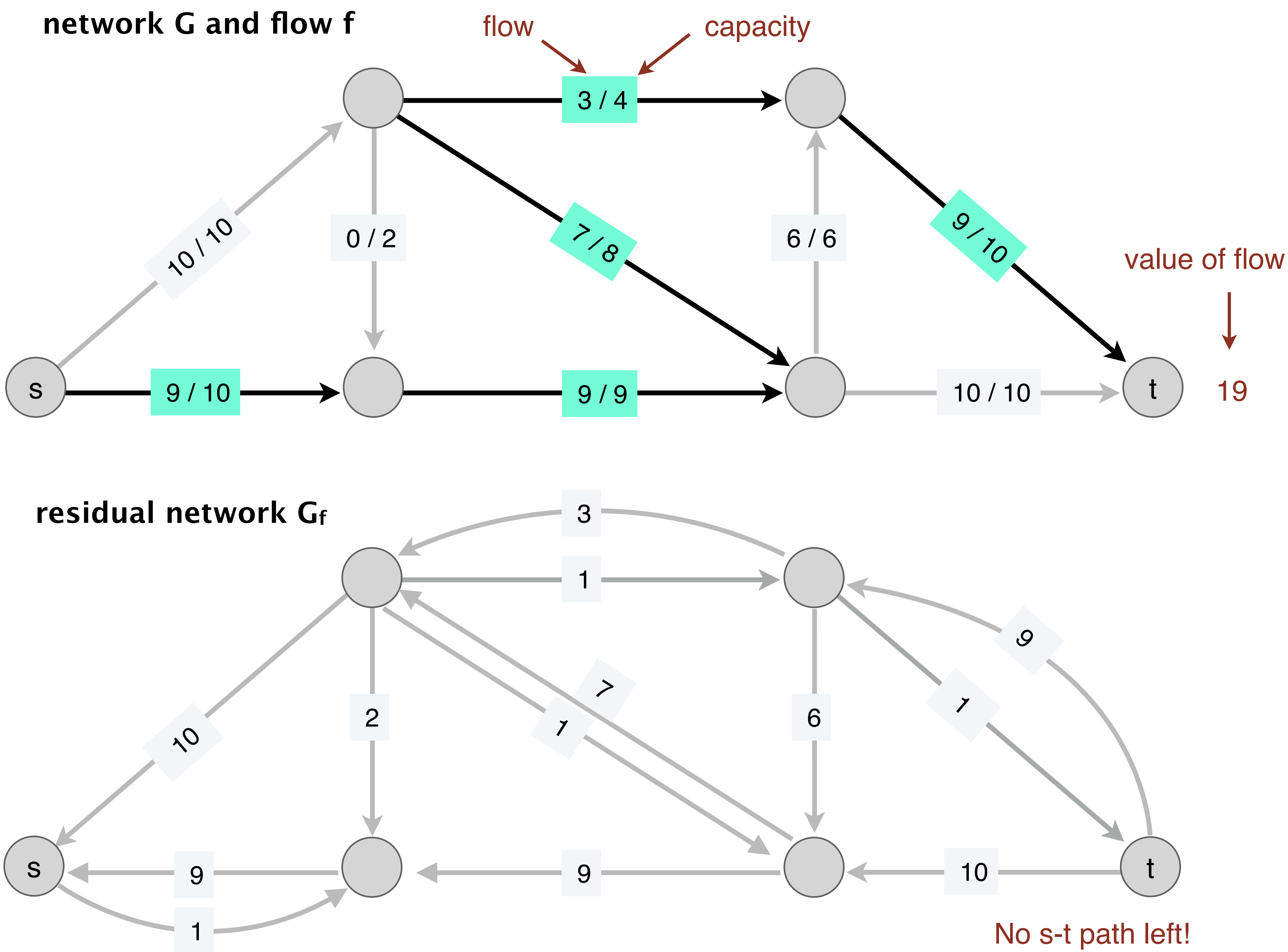
network G and flow f



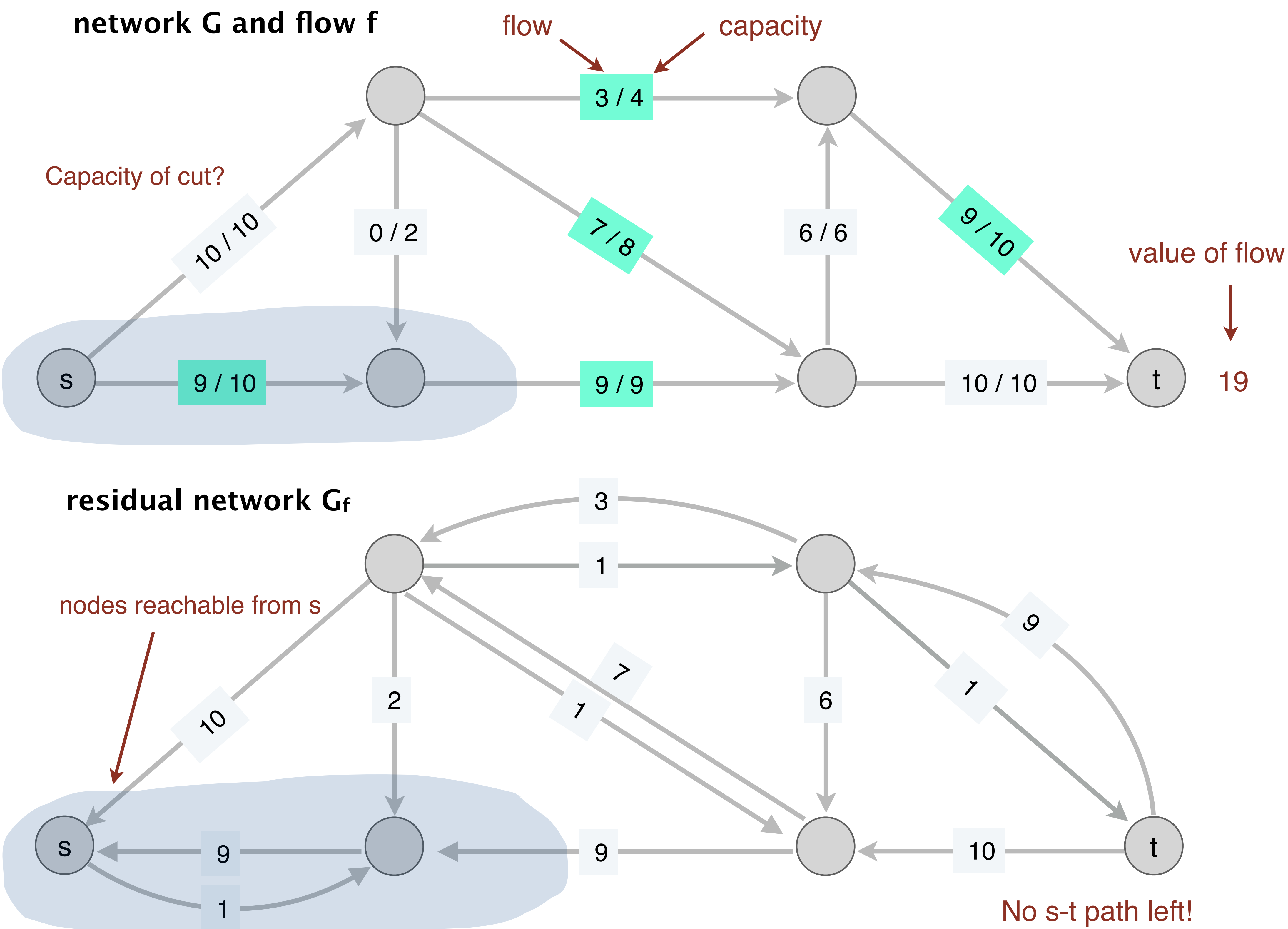
P in residual network G_f



Ford-Fulkerson Example



Ford-Fulkerson Example



Analysis: Ford-Fulkerson

Analysis Outline

- Feasibility and value of flow:
  - Show that each time we update the flow, we are routing a feasible s-t flow through the network
  - And that value of this flow increases each time by that amount
- Optimality:
  - Final value of flow is the maximum possible
- Running time:
  - How long does it take for the algorithm to terminate?
- Space:
  - How much total space are we using

Feasibility of Flow

- **Claim**. Let f be a feasible flow in G and let P be an augmenting path in G_f with bottleneck capacity b. Let $f' \leftarrow$ AUGMENT(f, P), then f' is **a feasible flow**.
- **Proof**. Only need to verify constraints on the edges of P (since $f' = f$ for other edges). Let $e = (u, v) \in P$
  - If e is a forward edge: $f'(e) = f(e) + b$

    $$\leq f(e) \; + \; (c(e) - f(e)) \; = \; c(e)$$
  - If e is a backward edge: $f'(e) = f(e) - b$

    $$\geq f(e) \; - \; f(e) \; = \; 0$$
- Conservation constraint hold on any node in $u \in P$:
  - $f_{in}(u) = f_{out}(u)$, therefore $f'_{in}(u) = f'_{out}(u)$ for both cases

Value of Flow: Making Progress

- **Claim**. Let f be a feasible flow in G and let P be an augmenting path in G_f with bottleneck capacity b. Let $f' \leftarrow$ AUGMENT(f, P), then $v(f') = v(f) + b$.
- **Proof**.
    - First edge $e \in P$ must be out of s in G_f
    - (P is simple so never visits s again)
    - e must be a forward edge (P is a path from s to t)
    - Thus $f(e)$ increases by b, increasing $v(f)$ by b ∎
- Note. Means the algorithm makes forward progress each time!

Optimality

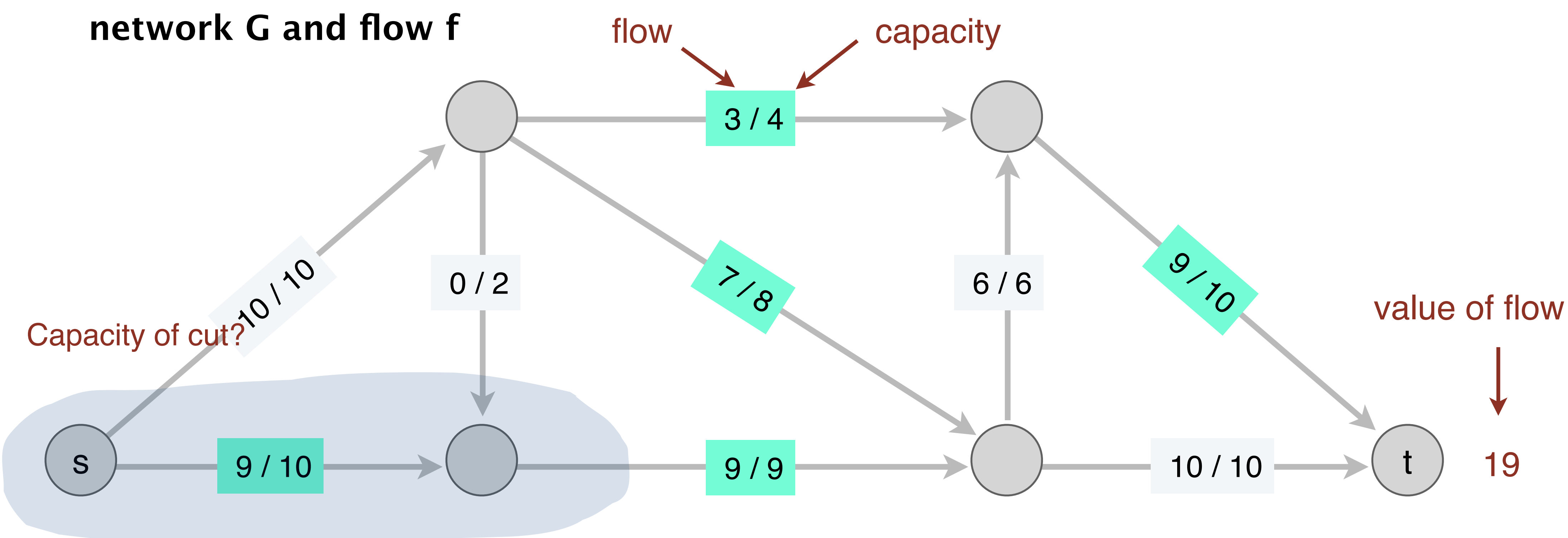
Ford-Fulkerson Optimality

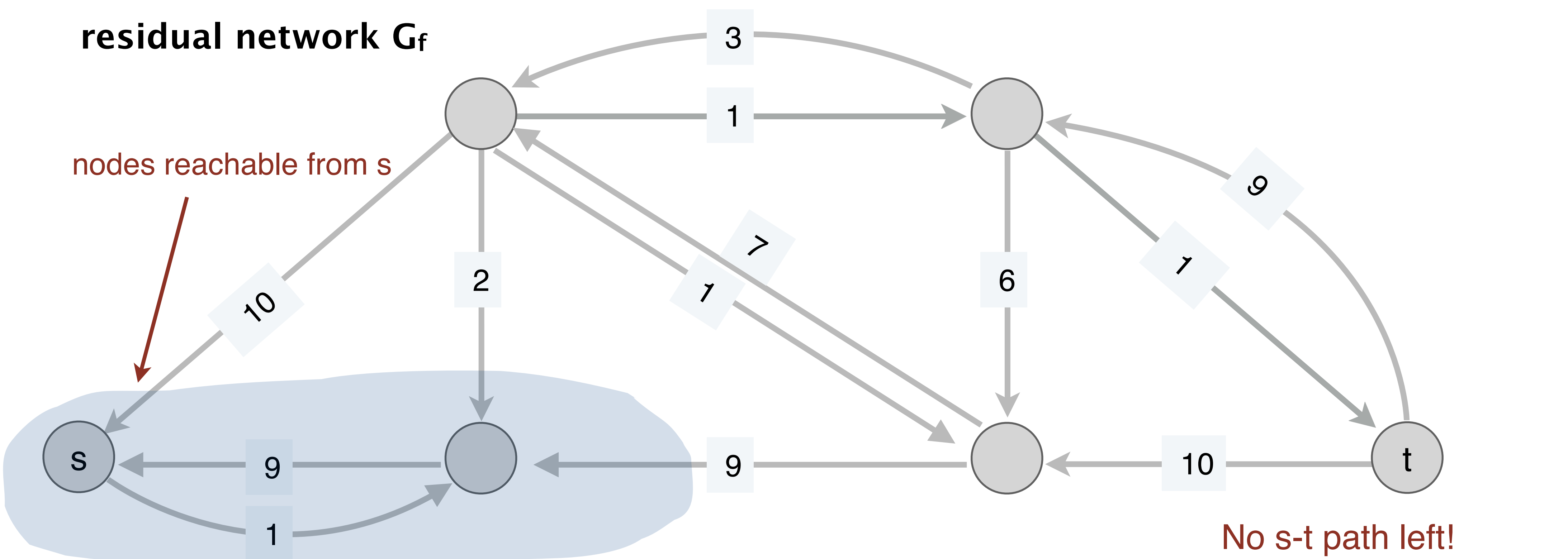
- **Recall**: If f is any feasible s-t flow and (S, T) is any s-t cut then $v(f) \leq c(S, T)$.
- We will show that the Ford-Fulkerson algorithm terminates in a flow that achieves equality, that is,
- Ford-Fulkerson finds a flow f^* and there exists a cut (S^*, T^*) such that, $v(f^*) = c(S^*, T^*)$
- Proving this shows that it finds the maximum flow (and the min cut)
- This also **proves the max-flow min-cut theorem**

Ford-Fulkerson Optimality

- **Lemma**. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f, then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- **Proof**.
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V - S^*$
- Is this an s-t cut?
  - $s \in S, t \in T$, $S \cup T = V$ and $S \cap T = \emptyset$
- Consider an edge $e = u \to v$ with $u \in S^*, v \in T^*$, then what can we say about $f(e)$?

Recall: Ford-Fulkerson Example





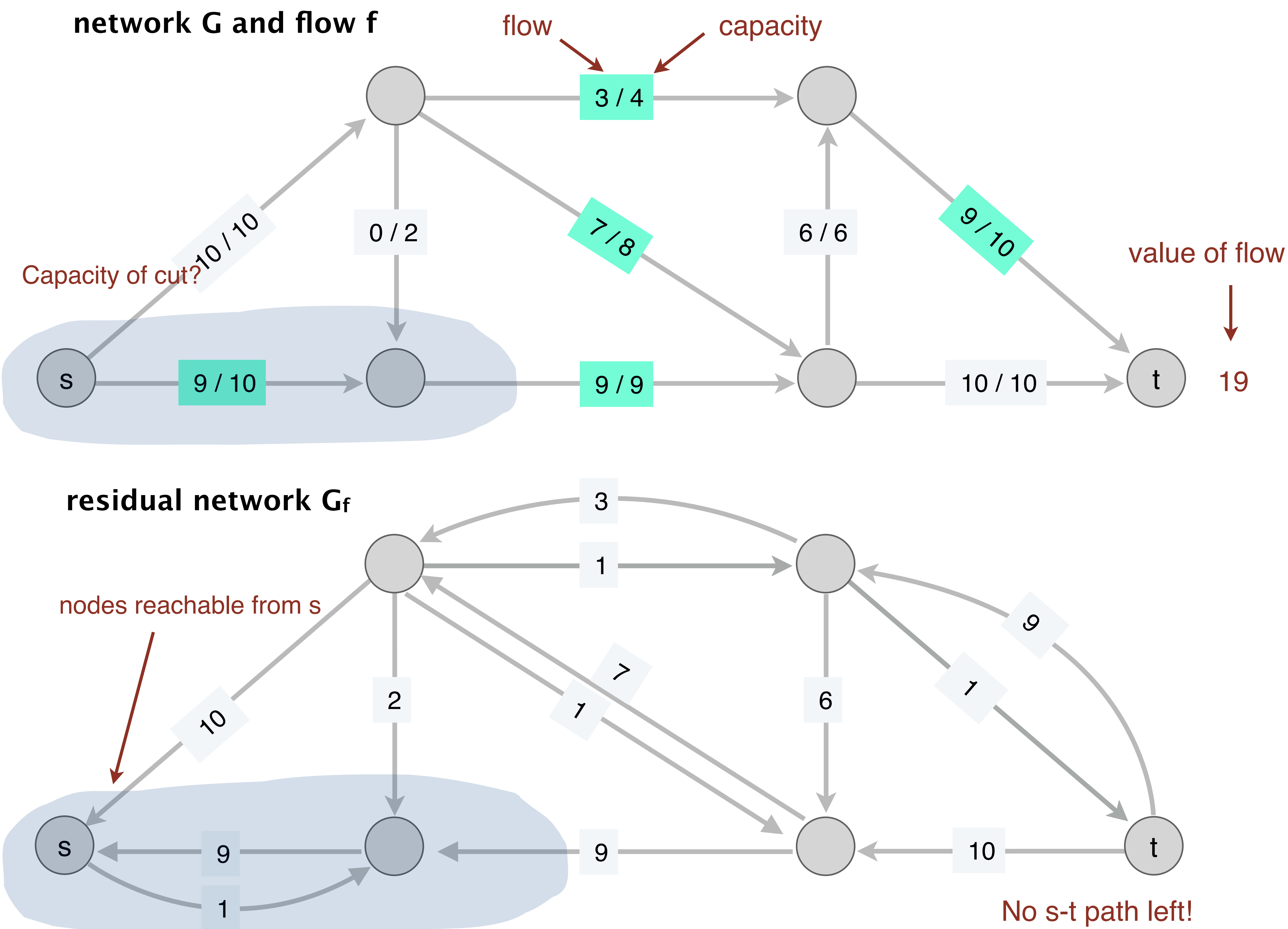
Ford-Fulkerson Optimality

- **Lemma**. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f, then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- **Proof**.
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V - S^*$
- Is this an s-t cut?
  - $s \in S, t \in T$, $S \cup T = V$ and $S \cap T = \varnothing$
- Consider an edge $e = u \rightarrow v$ with $u \in S^*, v \in T^*$, then what can we say about $f(e)$?
  - $f(e) = c(e)$

Ford-Fulkerson Optimality

- **Lemma**. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f, then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- **Proof**. **(Cont.)**
- Let $S^* = \{v \mid v$ is reachable from s in $G_f\}$, $T^* = V - S^*$
- Is this an s-t cut?
  - $s \in S, t \in T$, $S \cup T = V$ and $S \cap T = \emptyset$
- Consider an edge $e = w \rightarrow v$ with $v \in S^*, w \in T^*$, then what can we say about $f(e)$?

Recall: Ford-Fulkerson Example



Ford-Fulkerson Optimality

- **Lemma**. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f, then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.
- **Proof**. **(Cont.)**
- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V - S^*$
- Is this an s-t cut?
  - $s \in S, t \in T$, $S \cup T = V$ and $S \cap T = \emptyset$
- Consider an edge $e = w \rightarrow v$ with $v \in S^*, w \in T^*$, then what can we say about $f(e)$?
  - $f(e) = 0$

Ford-Fulkerson Optimality

- **Lemma**. Let f be a s-t flow in G such that there is no augmenting path in the residual graph G_f, then there exists a cut (S^*, T^*) such that $v(f) = c(S^*, T^*)$.

- **Proof**. **(Cont.)**

- Let $S^* = \{v \mid v \text{ is reachable from } s \text{ in } G_f\}$, $T^* = V - S^*$

- Thus, all edges leaving S^* are completely saturated and all edges entering S^* have zero flow

- $v(f) = f_{out}(S^*) - f_{in}(S^*) = f_{out}(S^*) = c(S^*, T^*)$ ∎

- **Corollary**. Ford-Fulkerson returns the maximum flow.

Ford-Fulkerson Algorithm
Running Time

Ford-Fulkerson Performance

FORD–FULKERSON(G)

FOREACH edge $e \in E$: $f(e) \leftarrow 0$.

$G_f \leftarrow$ residual network of G with respect to flow f.

WHILE (there exists an s⇝t path P in G_f)

    $f \leftarrow$ AUGMENT(f, P).

    Update G_f.

RETURN f.

- Does the algorithm terminate?
- Can we bound the number of iterations it does?
- Running time?

Ford-Fulkerson Running Time

- Recall we proved that with each call to AUGMENT, we increase **value of flow** by $b = \text{bottleneck}(G_f, P)$
- **Assumption**. Suppose all capacities $c(e)$ are integers.
- **Integrality invariant.** Throughout Ford–Fulkerson, every edge flow $f(e)$ and corresponding residual capacity is an integer. Thus $b \geq 1$.
- Let $C = \max_u c(s \to u)$ be the maximum capacity among edges leaving the source s.
- It must be that $v(f) \leq (n-1)C$
- Since, $v(f)$ increases by $b \geq 1$ in each iteration, it follows that FF algorithm terminates in at most $v(f) = O(nC)$ iterations.

Ford-Fulkerson Performance

```
FORD–FULKERSON(G)
─────────────────────────────────────────────────────────────
FOREACH edge e ∈ E : f(e) ← 0.
G_f ← residual network of G with respect to flow f.
WHILE (there exists an s⇝t path P in G_f)
    f ← AUGMENT(f, P).
    Update G_f.
RETURN f.
```

- Operations in each iteration?
  - Find an augmenting path in G_f
  - Augment flow on path
  - Update G_f

Ford-Fulkerson Running Time

- **Claim.** Ford-Fulkerson can be implemented to run in time $O(nmC)$, where $m = |E| \geq n - 1$ and $C = \max_u c(s \to u)$.
- **Proof**. Time taken by each iteration:
- Finding an augmenting path in G_f
  - G_f has at most $2m$ edges, using BFS/DFS takes $O(m + n) = O(m)$ time
- Augmenting flow in P takes $O(n)$ time
- Given new flow, we can build new residual graph in $O(m)$ time
- Overall, $O(m)$ time per iteration ∎