COMP 355
Advanced Algorithms

All-Pairs Shortest Paths
Floyd-Warshall Algorithm
Section 25.2 (CLRS): Not in KT
Network Flows:
Basics & Ford-Fulkerson Algorithm
Section 7.1-7.3 (KT)

Rhodes College
—1848—

All-Pairs Shortest Paths

- Generalization of single-source shortest path: computing shortest path between all pairs of vertices

- Let G = (V, E) be a directed graph with edge weights.

- Find the cost of the shortest path between all pairs of vertices in G.

Possible Algorithms

- If no negative weights:
  - Run Dijkstra's with each vertex as the source
  - Runtime: O(VE lg V) (if we use binary min-heap implementation)
- If negative weights, but no negative cycles:
  - Run Bellman-Ford algorithm once from each vertex
  - Runtime: $O(V^2E)$ (on a dense graph = $O(V^4)$)
- Can we do better (assuming negative edges)?
  - Yes! $O(V^3)$ using dynamic programming

Input/Output

- Input Format:
  - input is an *n x n* matrix *w* of edge weights, which are based on the edge weights in the digraph.
  - We let w_{ij} denote the entry in row *i* and column *j* of *w*.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

- Output Format:
  - n x n distance matrix D = d_{ij} where d_{ij} = δ(i, j), the shortest path from vertex i to vertex j.
  - To recover the actual shortest path, we can compute an auxillary matrix mid[i, j] where the value of mid[i, j] will be a vertex that is somewhere along the path from i to j. (null if no such vertex exists)

Observations

- A shortest path does not contain the same vertex more than once.
- For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, ..., k\}$, there are two possibilities:
  - 1. k is not a vertex on the path,
    so the shortest such path has length d_{ij}^{k-1}
  - 2. k is a vertex on the path,
    so the shortest such path is $d_{ik}^{k-1} + d_{kj}^{k-1}$
- So we see that we can recursively define $d_{ij}^{(k)}$ as

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1 \end{cases}$$

Floyd-Warshall Algorithm

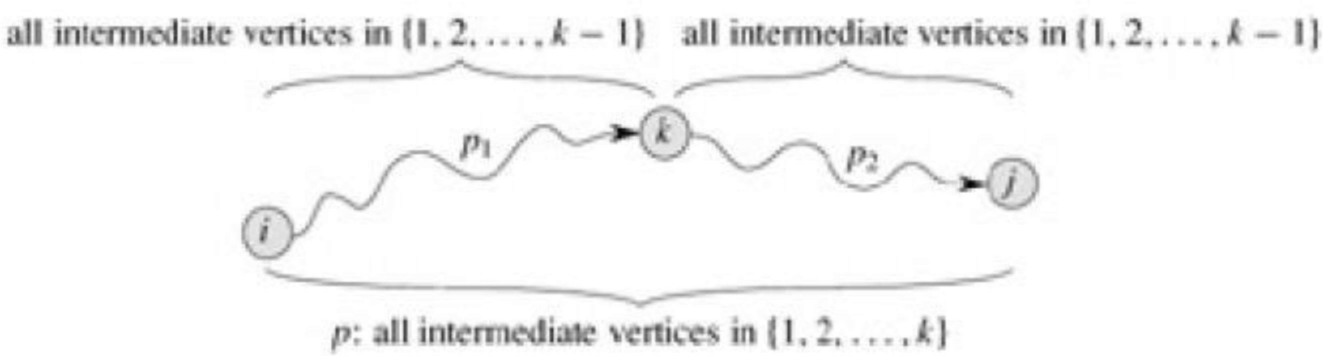

```
Floyd_Warshall(int n, int w[1..n, 1..n]) {
    array d[1..n, 1..n]
    for i = 1 to n do {                         // initialize
        for j = 1 to n do {
            d[i,j] = W[i,j]
            mid[i,j] = null
        }
    }
    for k = 1 to n do                           // use intermediates {1..k}
        for i = 1 to n do                       // ...from i
            for j = 1 to n do                   // ...to j
                if (d[i,k] + d[k,j]) < d[i,j]) {
                    d[i,j] = d[i,k] + d[k,j]    // new shorter path length
                    mid[i,j] = k                // new path is through k
                }
    return d                                    // matrix of distances
}
```

Running Time: $\Theta(n^3)$
Space Required: $\Theta(n^2)$

Proof of Correctness

Inductive Hypothesis

Suppose that prior to the kth iteration it holds that for $i, j \in V$, d_{ij} contains the length of the shortest path Q from i to j in G containing only vertices in the set $\{1, 2, ..., k-1\}$, and π_{ij} contains the immediate predecessor of j on path Q.



Network Flows

Def. Name of a variety of related graph optimization problems

Given a flow network, which is essentially a directed graph with nonnegative edge weights.

- Think of the edges as “pipes” that are capable of carrying some sort of “stuff.”
- Each edge of the network has a given *capacity*
- How much flow we can push from a designated source node to a designated sink node?

Maximum Flow and Minimum Cut

Max flow and min cut.

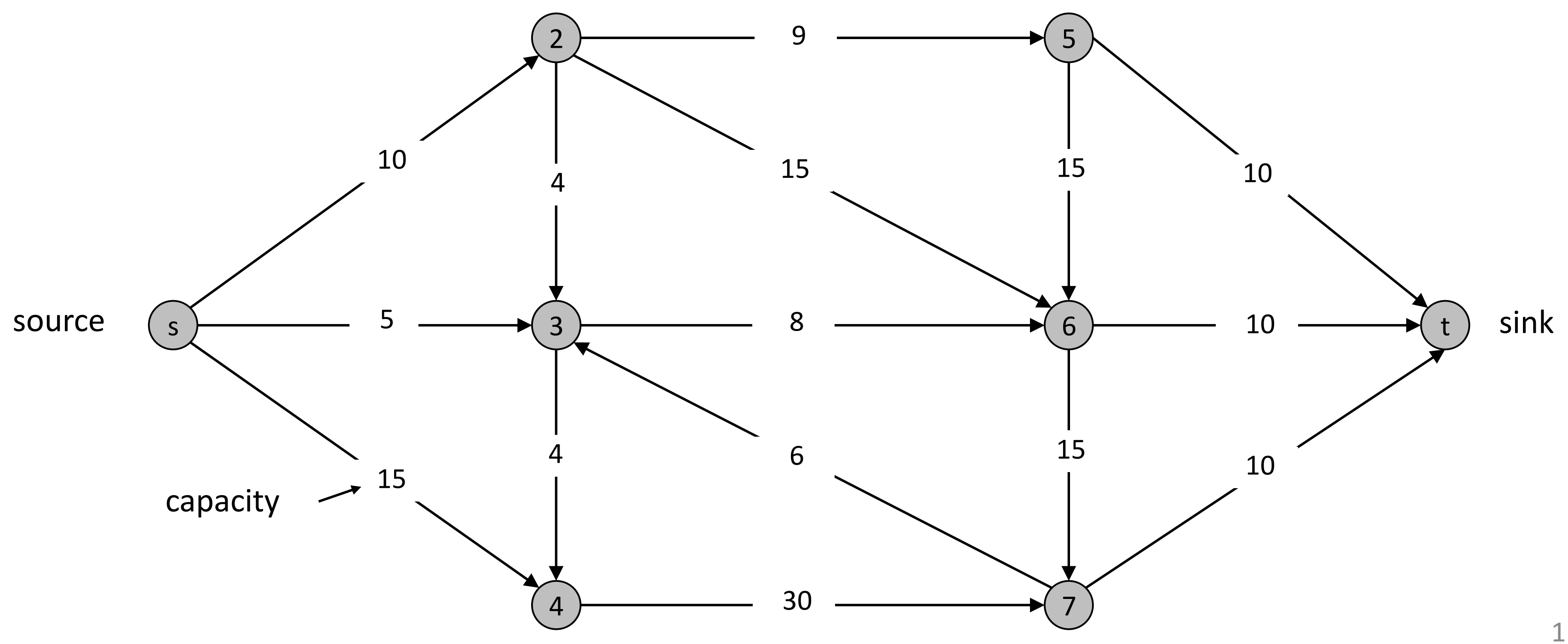
- Two very rich algorithmic problems.
- Cornerstone problems in combinatorial optimization.
- Beautiful mathematical duality.

Nontrivial applications / reductions.

- Data mining.
- Open-pit mining.
- Project selection.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Network reliability.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Network intrusion detection.
- Multi-camera scene reconstruction.
- And many more . . .

Flow Network

- Abstraction for material flowing through the edges.
- G = (V, E) directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- c(e) = capacity of edge e. (non-negative)



Flows, Capacities, and Conservation

Given an s-t network, a *flow* is a function f that maps each edge to a nonnegative real number and satisfies the following properties:

- **Capacity Constraint**: For all $e \in E, f(u,v) \leq c(u,v) = c(e)$
- **Flow conservation (or flow balance)**: For all $v \in V \ \{s,t\}$, the sum of flow along edges into v equals the sum of flows along edges out of v. $f^{in}(v) = f^{out}(v)$

If edge (u,v) not in E, then $f(u,v) = 0$

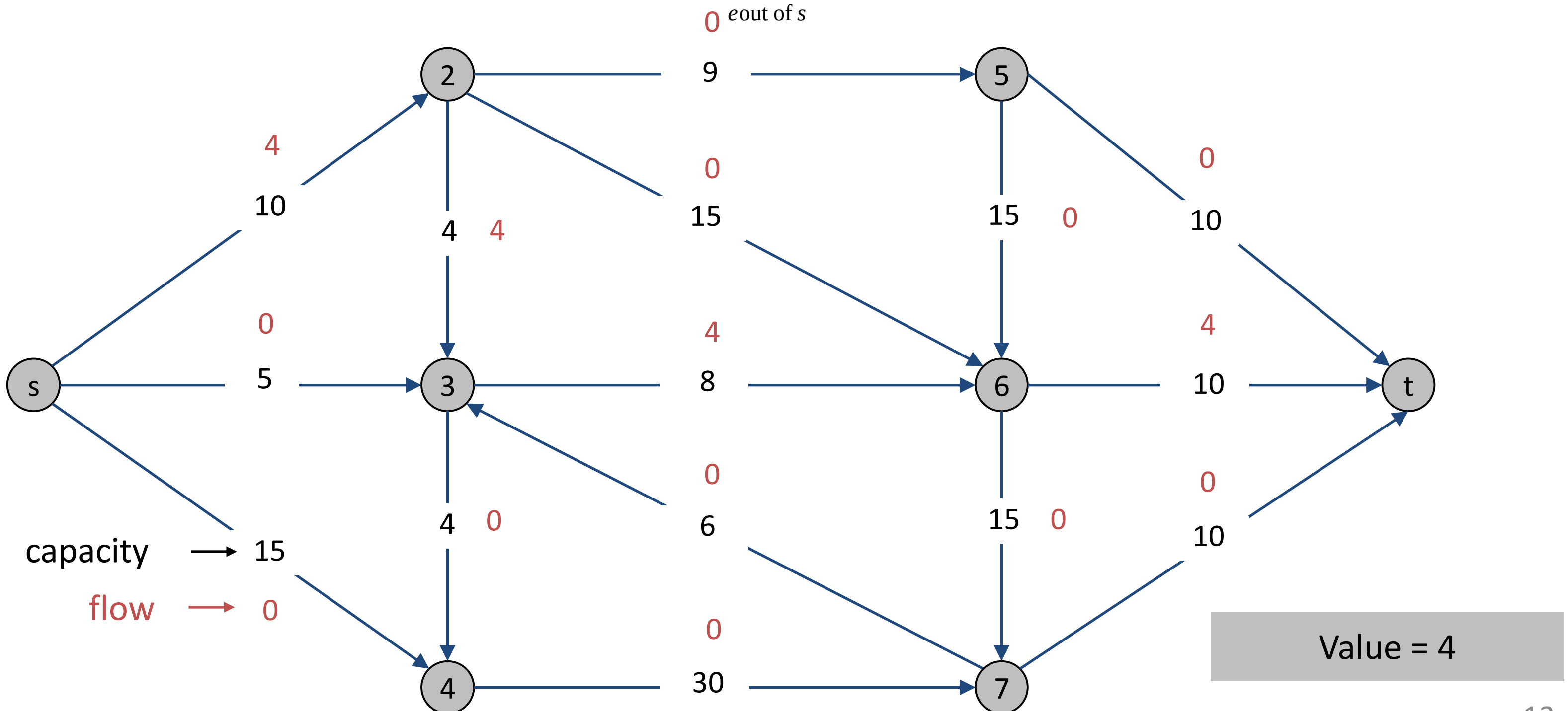
$$f^{in}(v) = \sum_{u \in V} f(u,v) \qquad f^{out}(v) = \sum_{w \in V} f(v,w)$$

Flows

Def. An s-t flow is a function that satisfies:

- For each e ∈ E: $0 \leq f(e) \leq c(e)$ (capacity)
- For each v ∈ V – {s, t}: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

Def. The value of a flow f is: $v(f) = \sum_{e \text{ out of } s} f(e)$.

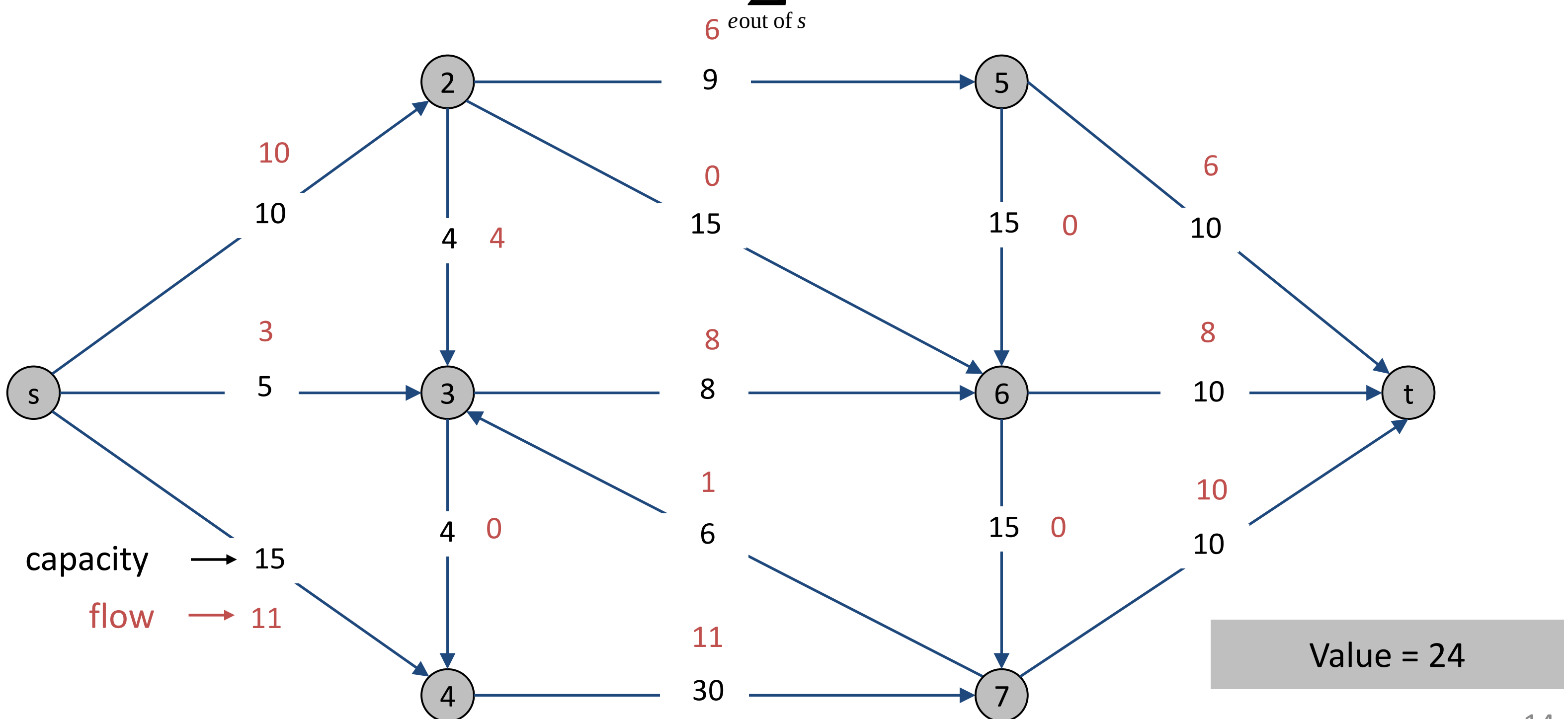


Flows

Def. An $s - t$ flow is a function that satisfies:

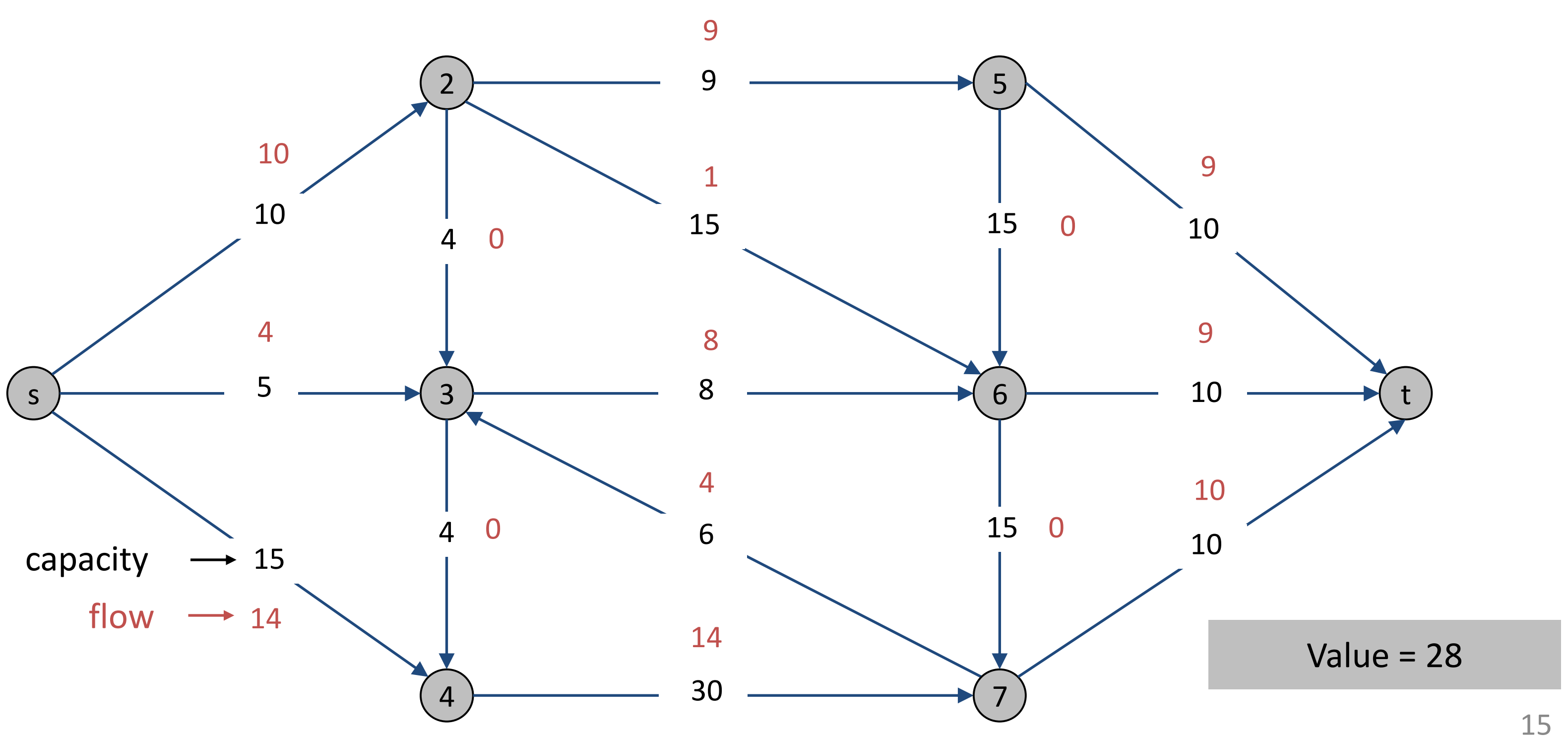
- For each $e \in E$: $0 \leq f(e) \leq c(e)$ (capacity)
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

Def. The value of a flow f is: $v(f) = \sum_{e \text{ out of } s} f(e)$.



Maximum Flow Problem

Max flow problem. Find s-t flow of maximum value.

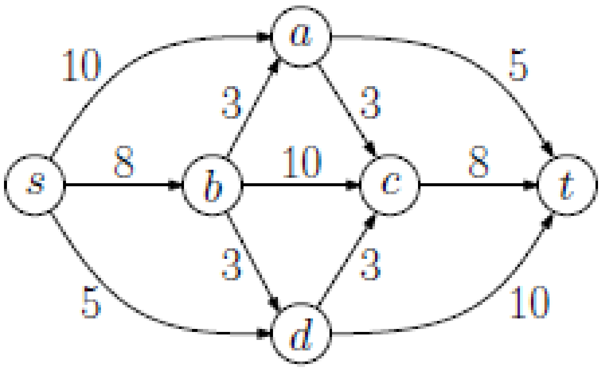


Path-Based Flows

Define an *s-t path* to be any simple path from s to t.

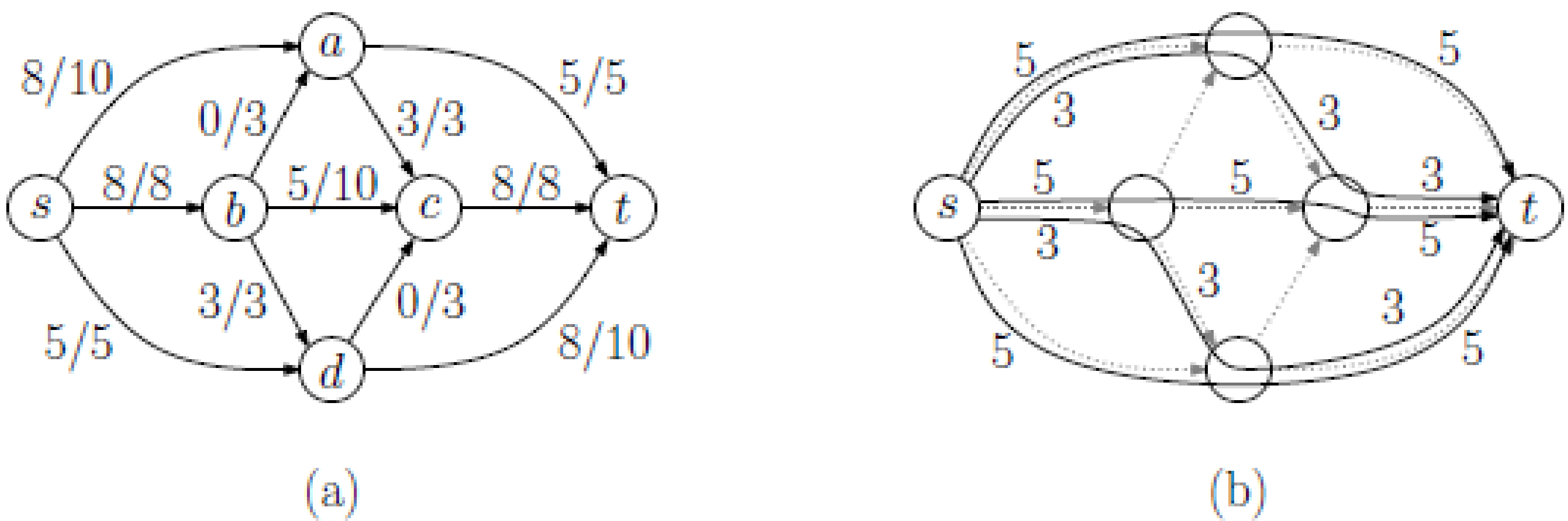
Ex. $\langle s,a,t\rangle$, $\langle s,b,a,c,t\rangle$ and $\langle s,d,c,t\rangle$ are all examples of $s-t$ paths.

Def. A *path-based flow* is a function that assigns each $s-t$ path a nonnegative real number such that, for every edge $(u,v) \in E$, the sum of the flows on all the paths containing this edge is at most $c(u,v)$.



Path-Based Flows

No need to provide a flow conservation constraint (each path that carries a flow into a vertex (excluding s and t), carries an equivalent amount of flow out of that vertex)



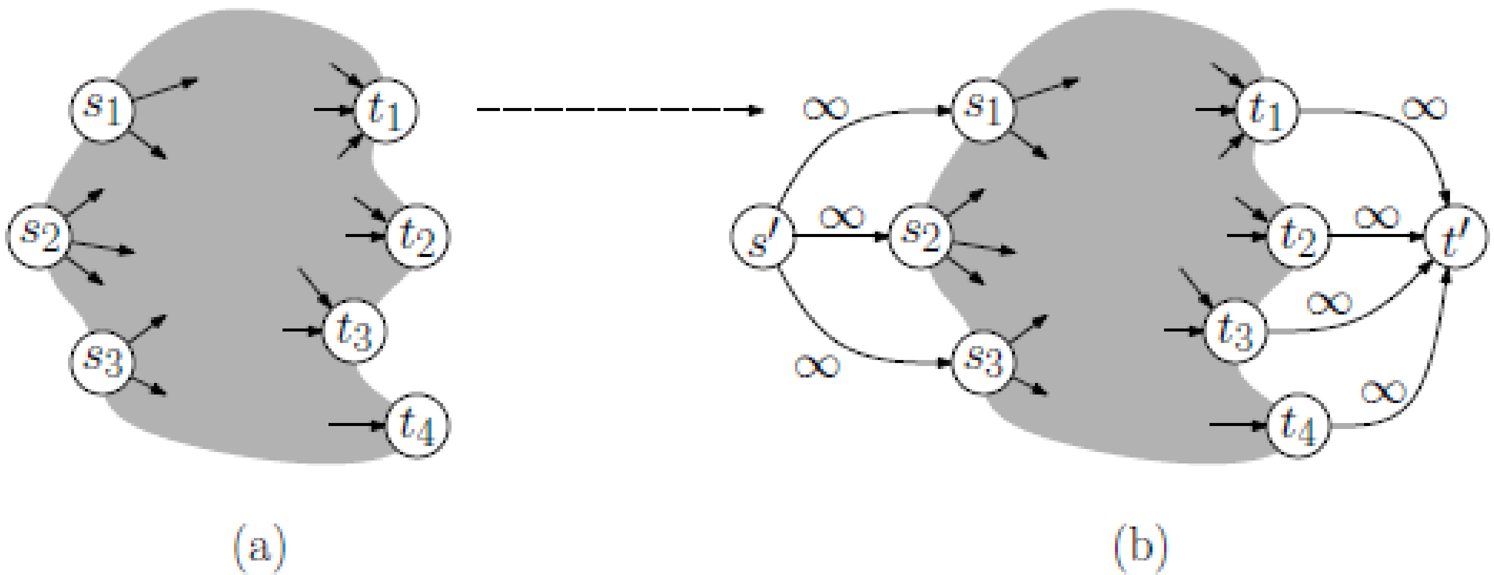
(a) An edge-based flow and (b) its path-based equivalent.

Path-Based Flows

Def. The *value* of a path-based flow is defined to be the total sum of all the flows on all the s-t paths of the network.

Claim: Given an s-t network G, under the assumption that there are no edges entering s or leaving t, G has an edge-based flow of value x if and only if G has a path-based flow of value x.

Multi-source, multi-sink networks

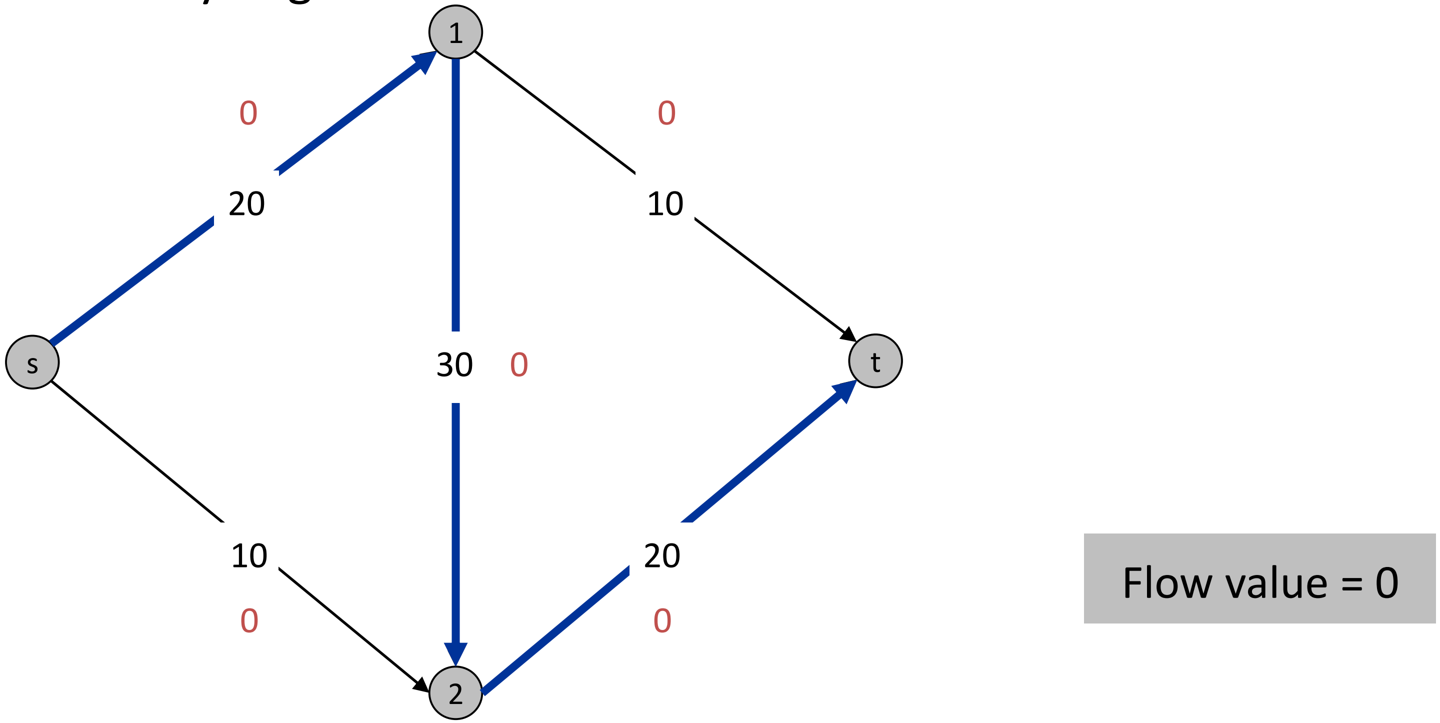


Reduction from (a) multi-source/multi-sink to
(b) single-source/single-sink.

Towards a Max Flow Algorithm

Greedy algorithm.

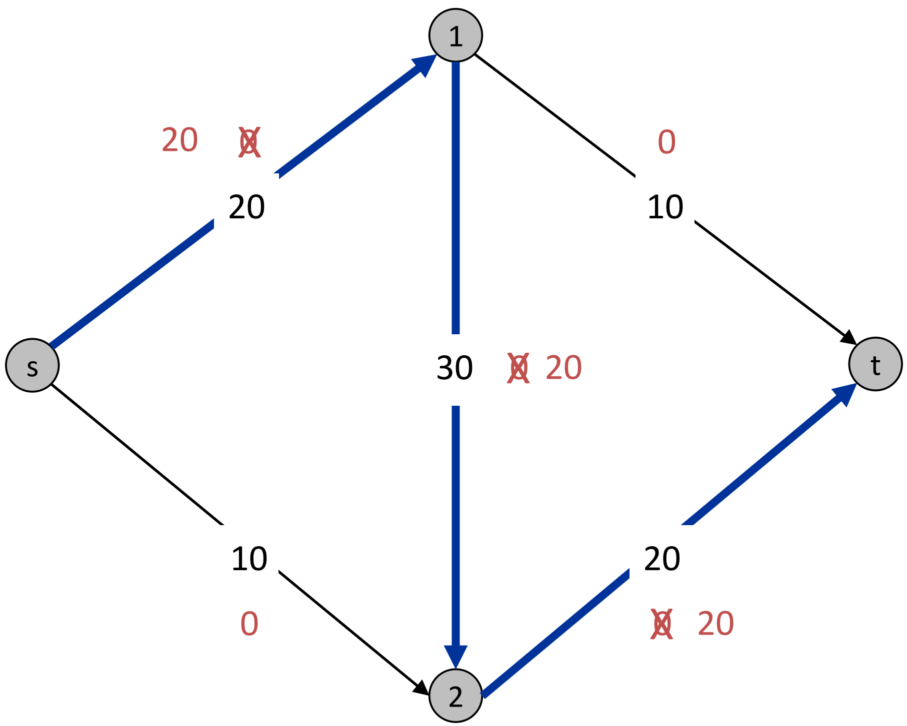
- Start with f(e) = 0 for all edge e $\in$ E.
- Find an s-t path P where each edge has f(e) < c(e).
- Augment flow along path P.
- Repeat until you get stuck.



Towards a Max Flow Algorithm

Greedy algorithm.

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



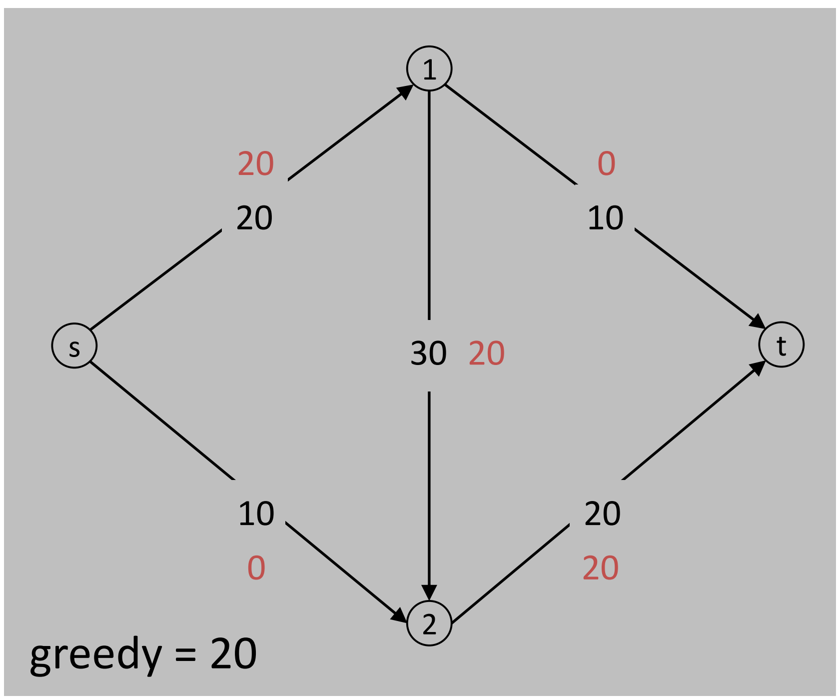
Flow value = 20

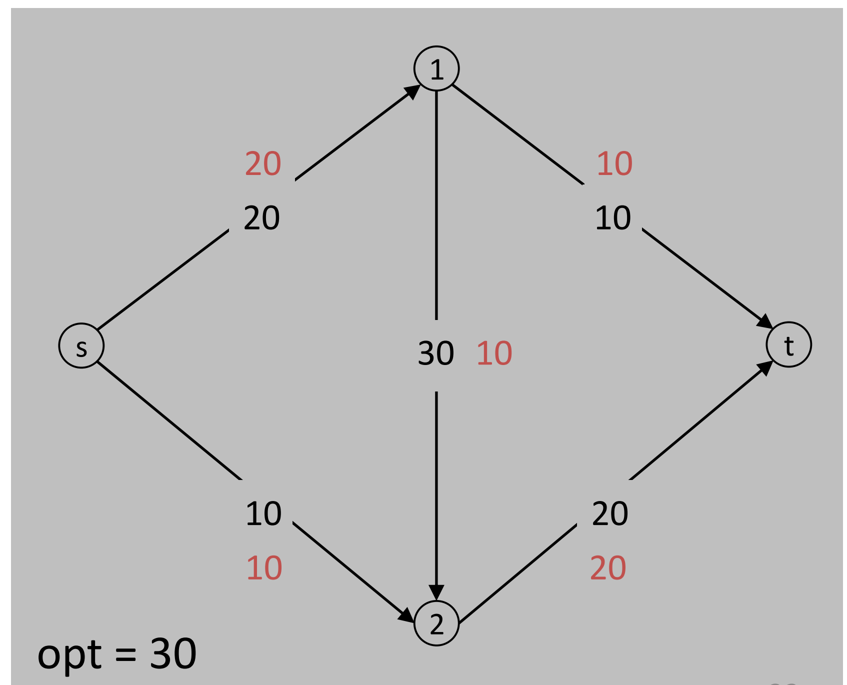
Towards a Max Flow Algorithm

Greedy algorithm.

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

locally optimality $\not\Rightarrow$ global optimality

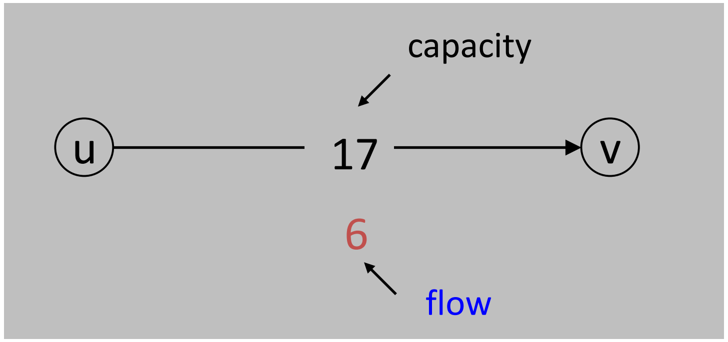




Residual Graph

Original edge: $e = (u, v) \in E$.

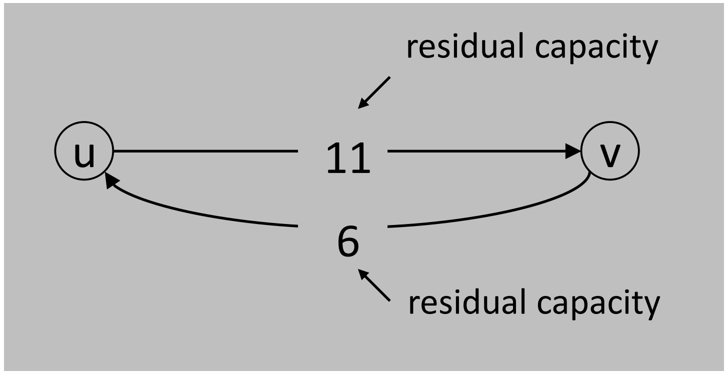
- Flow $f(e)$, capacity $c(e)$.



Residual edge.

- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:

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



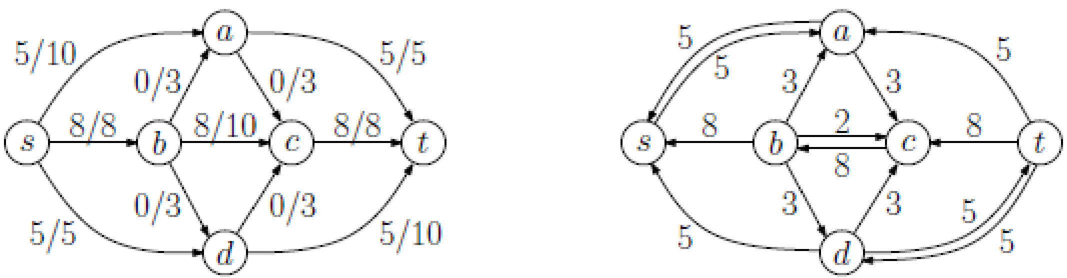
Residual graph: $G_f = (V, E_f)$.

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

Residual Graph

Forward edges: For each edge (u, v) for which $f(u, v) < c(u, v)$, create an edge (u, v) in G_f and assign it the capacity $c_f(u, v) = c(u, v) - f(u, v)$. Intuitively, this edge signifies that we can add up to $c_f(u, v)$ additional units of flow to this edge without violating the original capacity constraint.

Backward edges: For each edge (u, v) for which $f(u, v) > 0$, create an edge (v, u) in G_f and assign it a capacity of $c_f(v, u) = f(u, v)$. Intuitively, this edge signifies that we can cancel up to $f(u, v)$ units of flow along (u, v). Conceptually, by pushing positive flow along the reverse edge (v, u) we are decreasing the flow along the original edge (u, v).



(a): A flow f in network G (b): Residual network G_f

A flow f and the residual network G_f.

Augmenting Paths

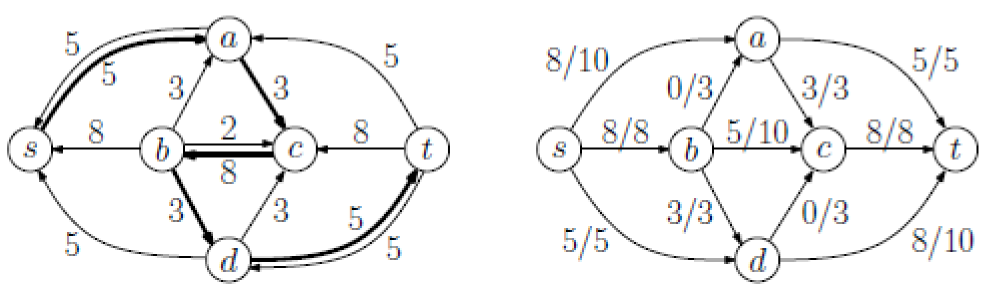
Consider a network G, let f be a flow in G, and let G_f be the associated residual network.

Def. An *augmenting path* is a simple path P from s to t in G_f .

Def. The *residual capacity* (also called the bottleneck capacity) of the path is the minimum capacity of any edge on the path. It is denoted $c_f\,(P)$.

Recall: all the edges of G_f are of strictly positive capacity, so $c_f\,(P)\ >\ 0.$

By pushing $c_f\,(P)$ units of flow along each edge of the path, we obtain a valid flow in G_{f}, and by the previous lemma, adding this to f results in a valid flow in G of strictly higher value.



(a): Augmenting path of capacity 3 (b): The flow after augmentation

Augmenting Path Algorithm

```
Augment(f, c, P) {
   b ← bottleneck(P)
   foreach e ∈ P {
      if (e ∈ E) f(e) ← f(e) + b
      else       f(eR) ← f(e) - b
   }
   return f
}
```

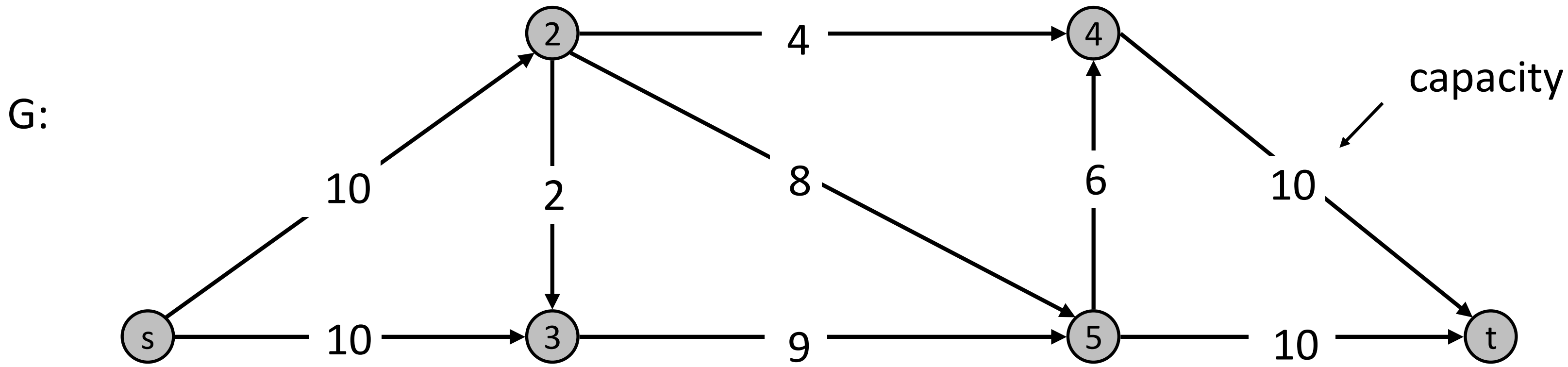
forward edge

reverse edge

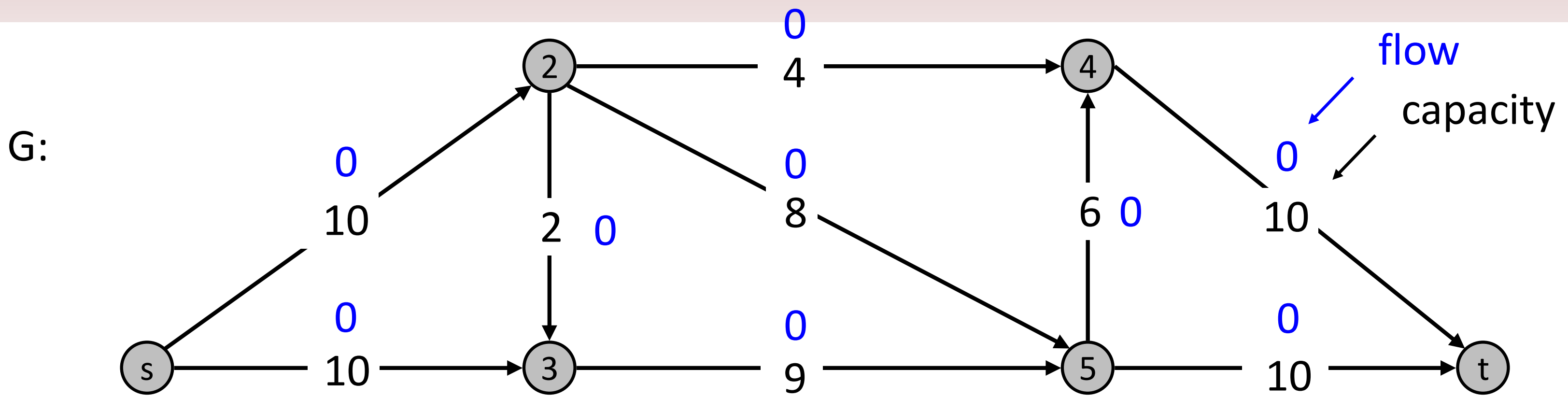
```
Ford-Fulkerson(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   Gf ← residual graph

   while (there exists augmenting path P) {
      f ← Augment(f, c, P)
      update Gf
   }
   return f
}
```

Ford-Fulkerson Algorithm



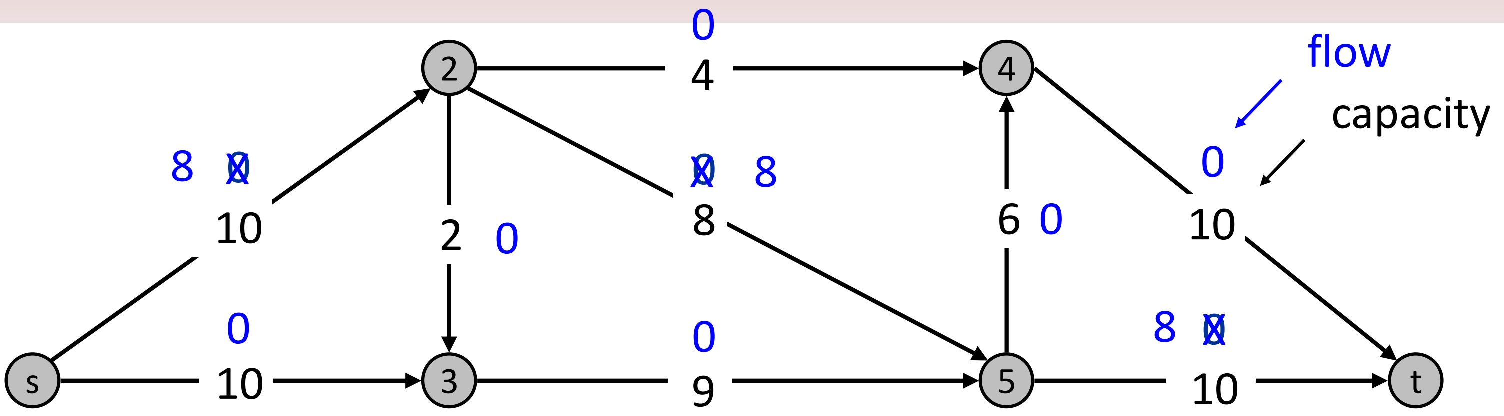
Ford-Fulkerson Algorithm



Flow value = 0

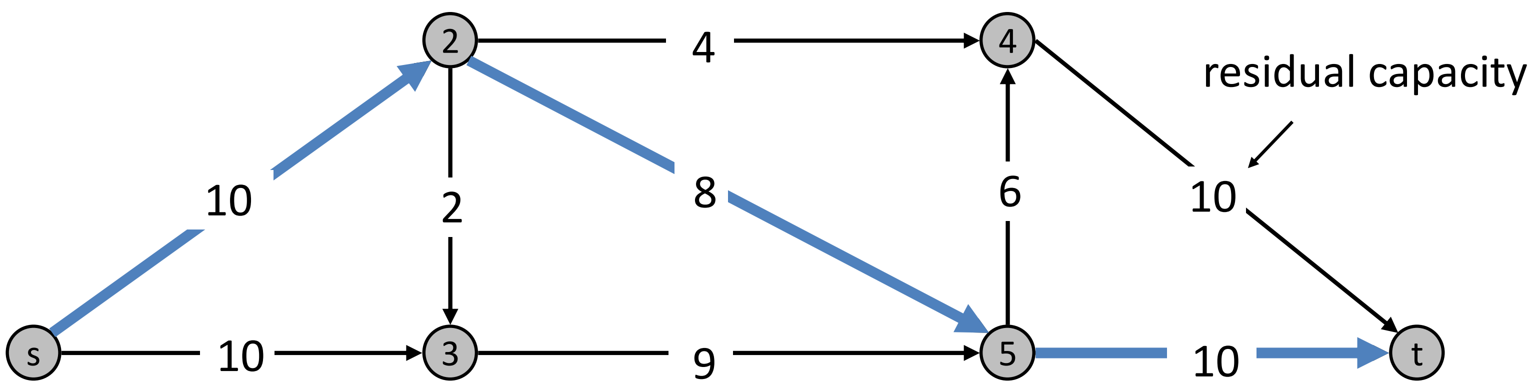
Ford-Fulkerson Algorithm

G:



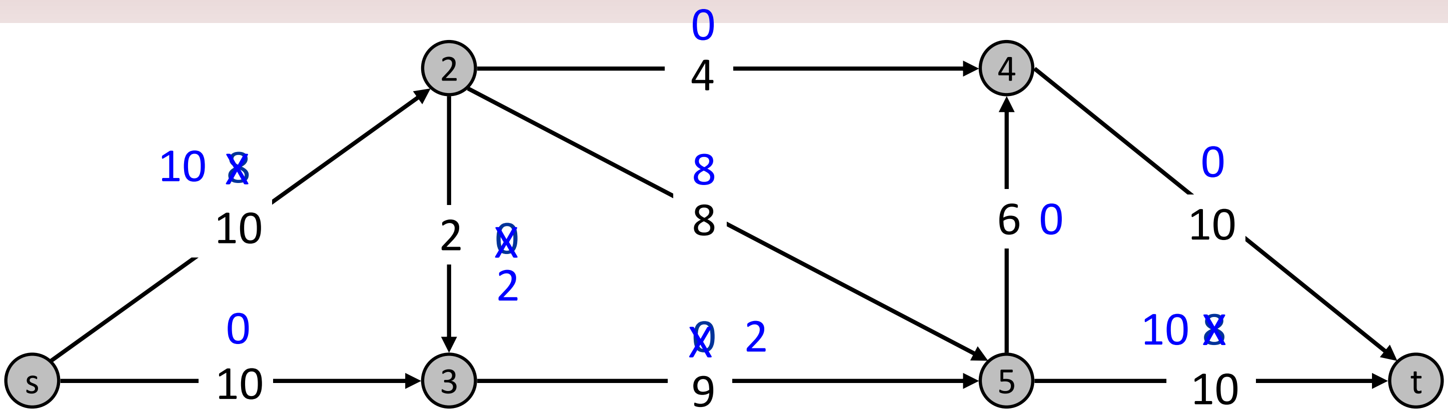
Flow value = 0

G_f:



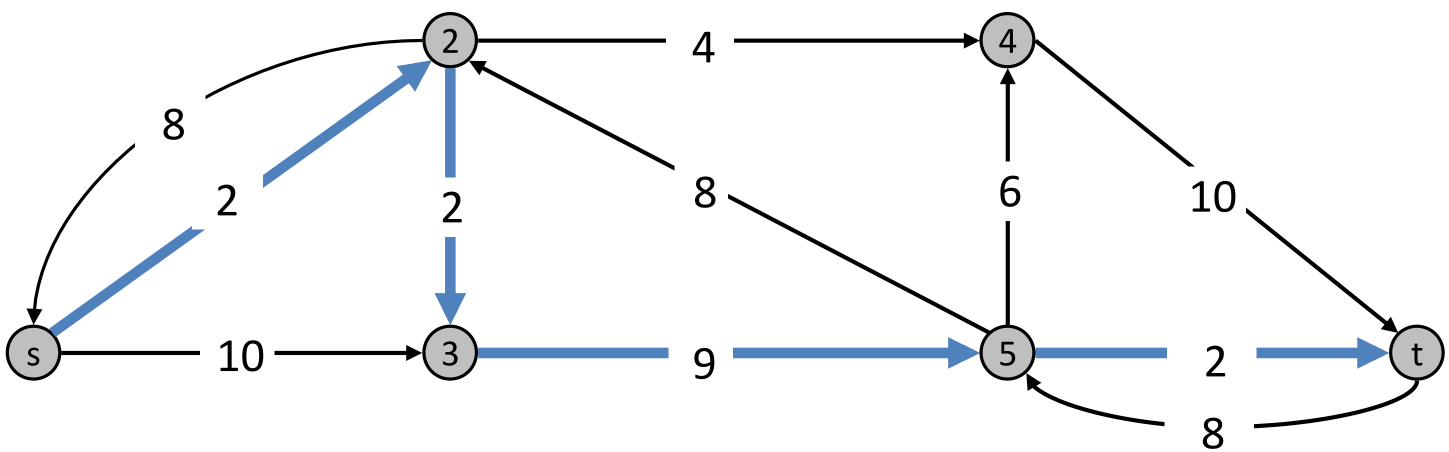
Ford-Fulkerson Algorithm

G:



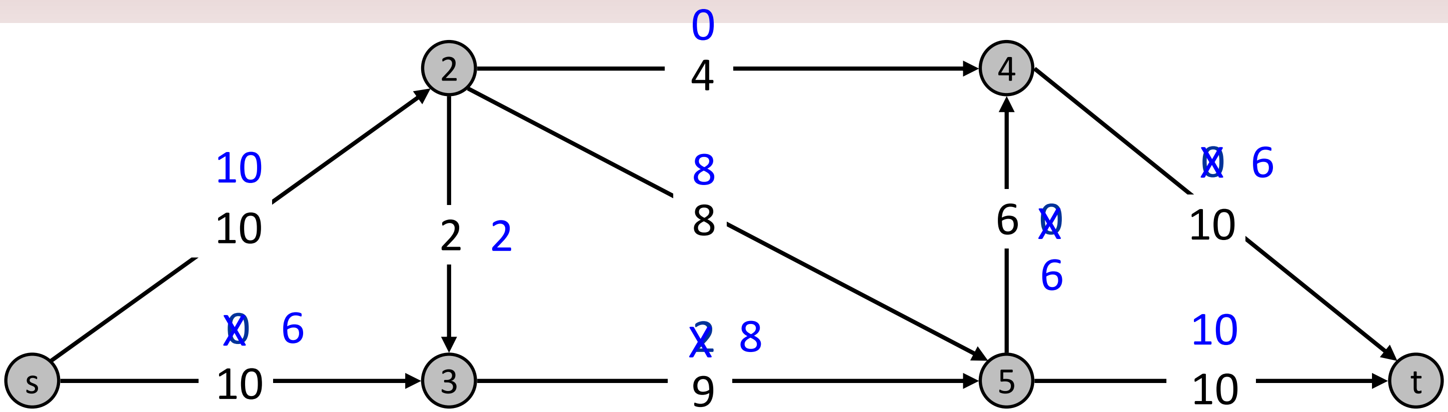
Flow value = 8

G_f:



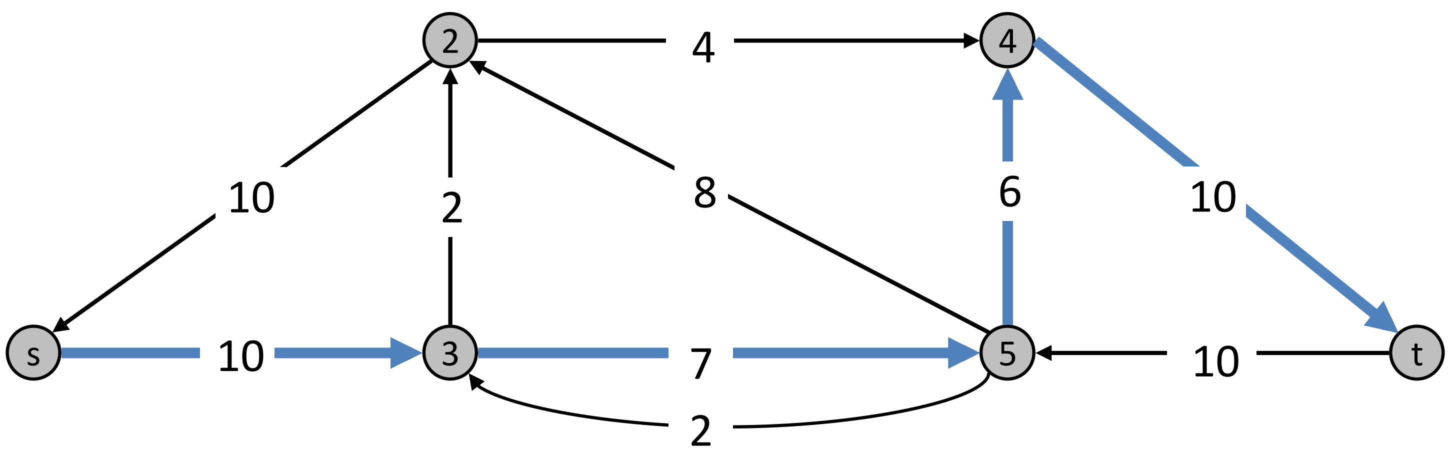
Ford-Fulkerson Algorithm

G:

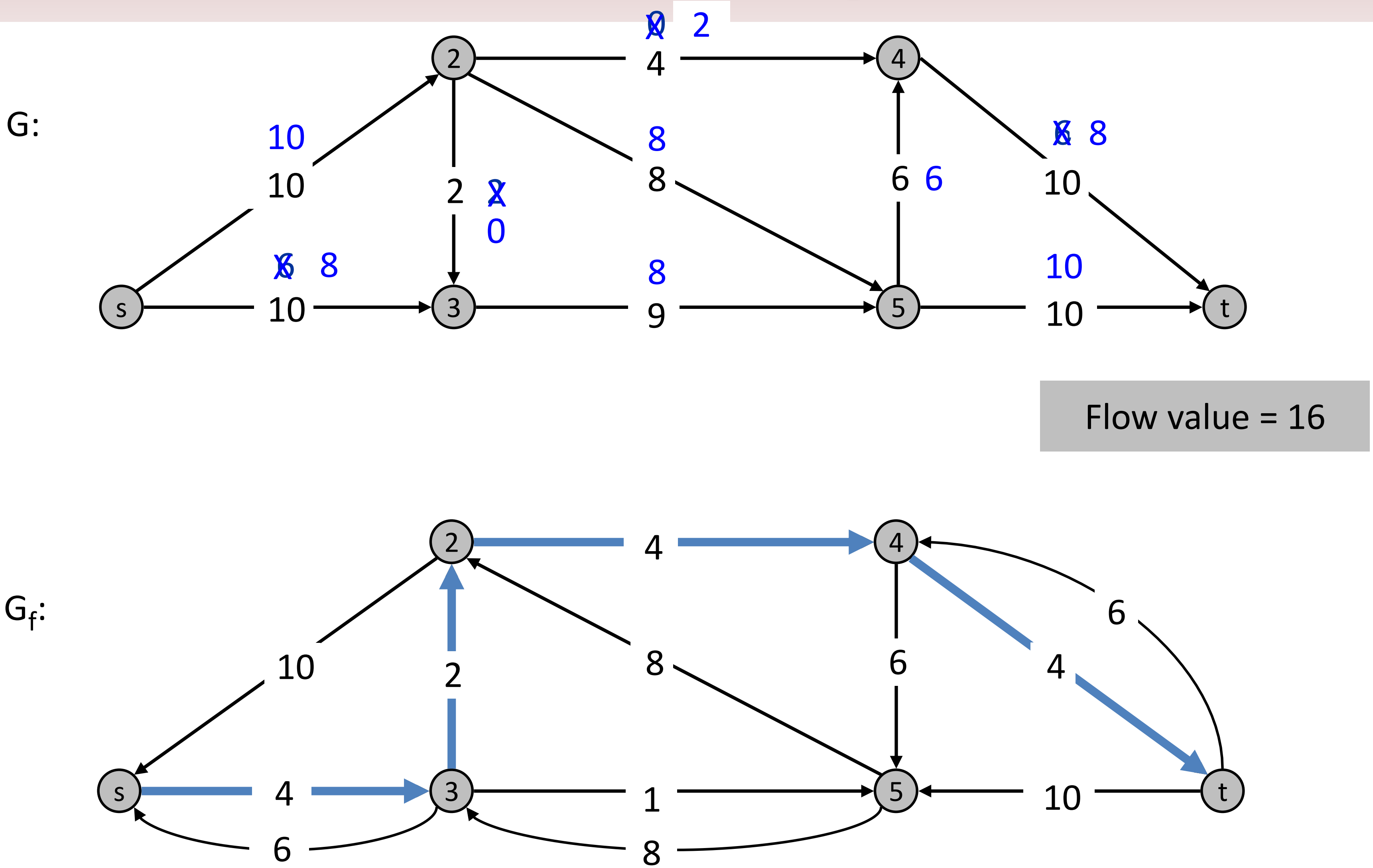


Flow value = 10

G_f:

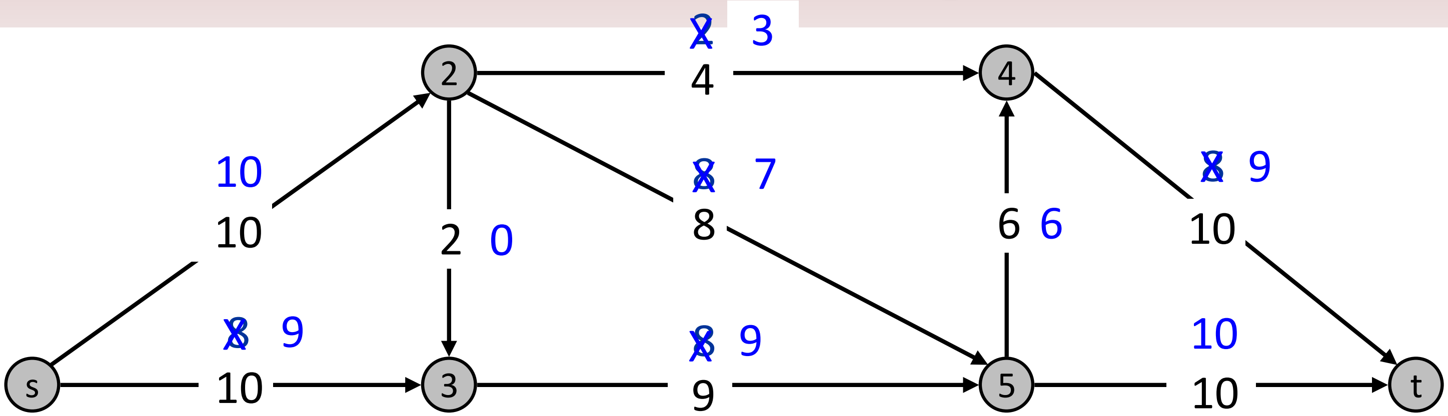


Ford-Fulkerson Algorithm



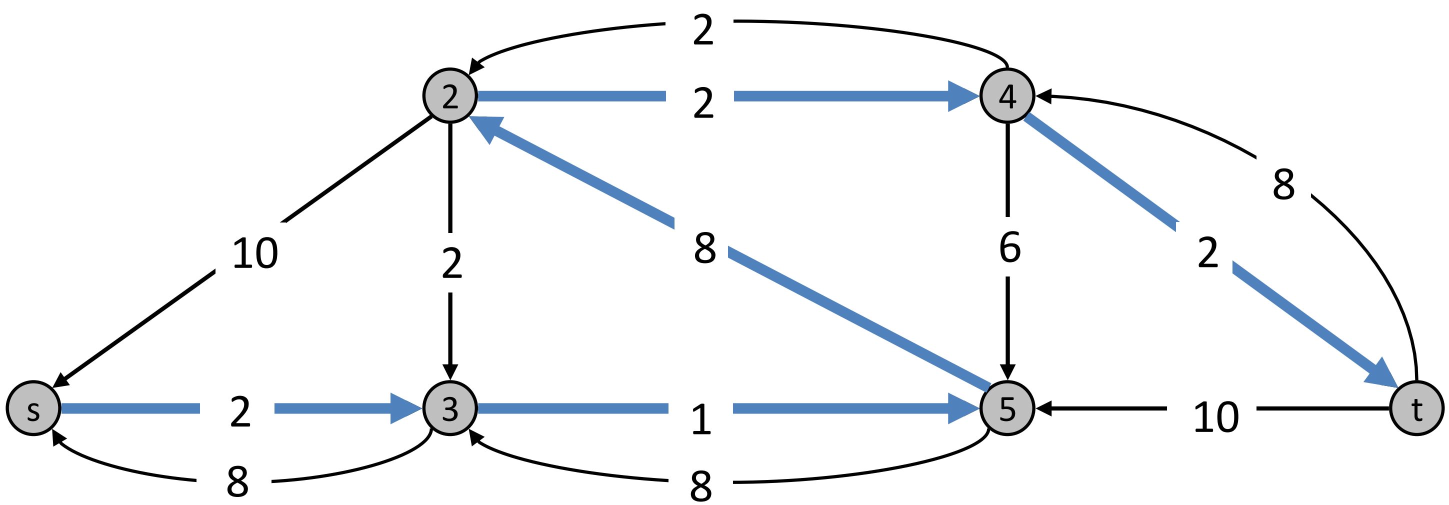
Ford-Fulkerson Algorithm

G:



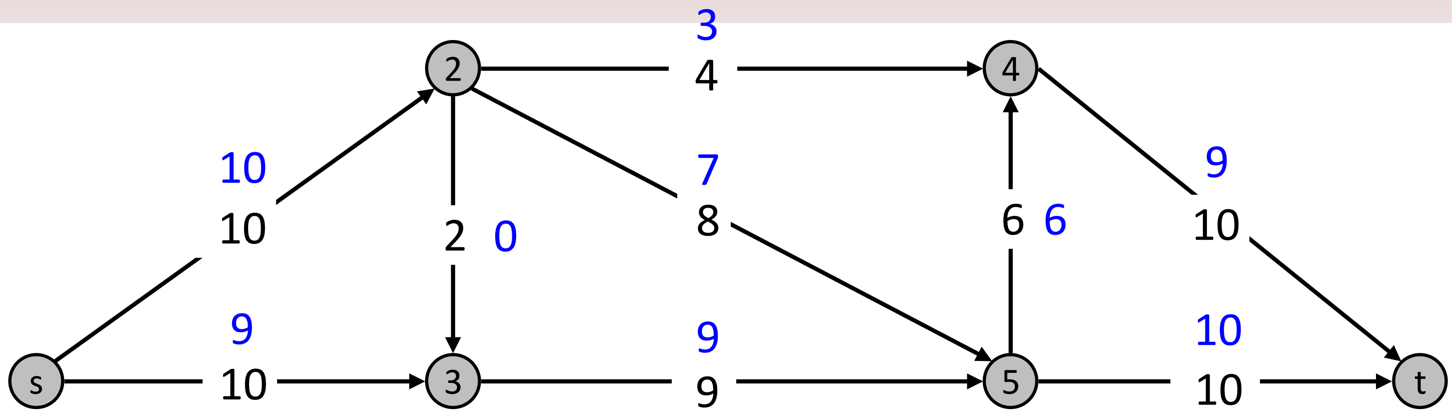
Flow value = 18

G_f:



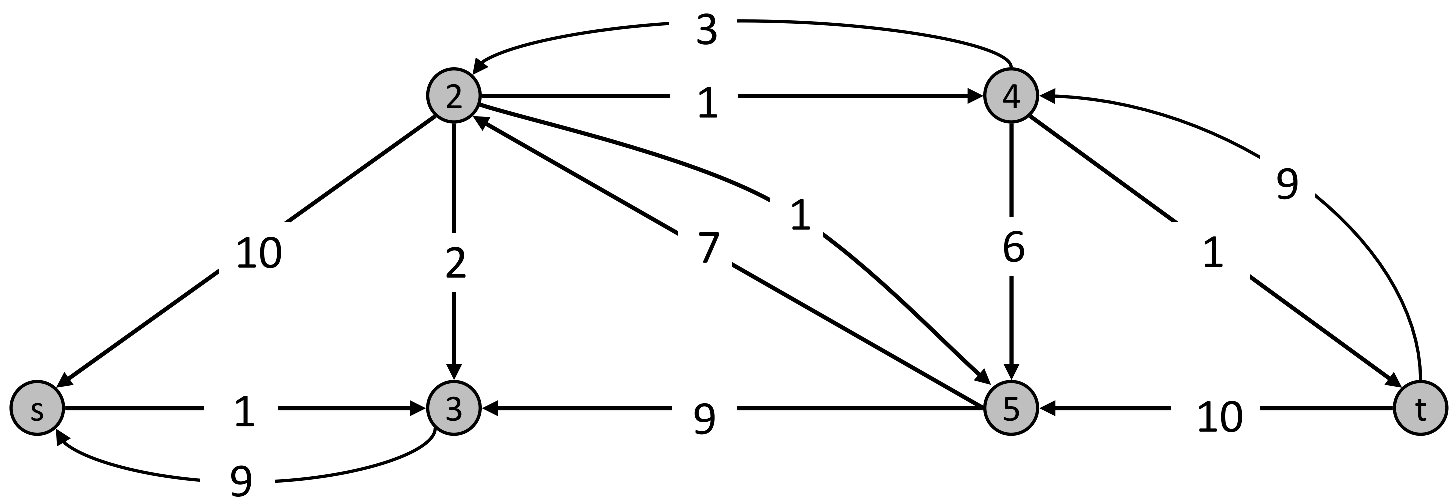
Ford-Fulkerson Algorithm

G:



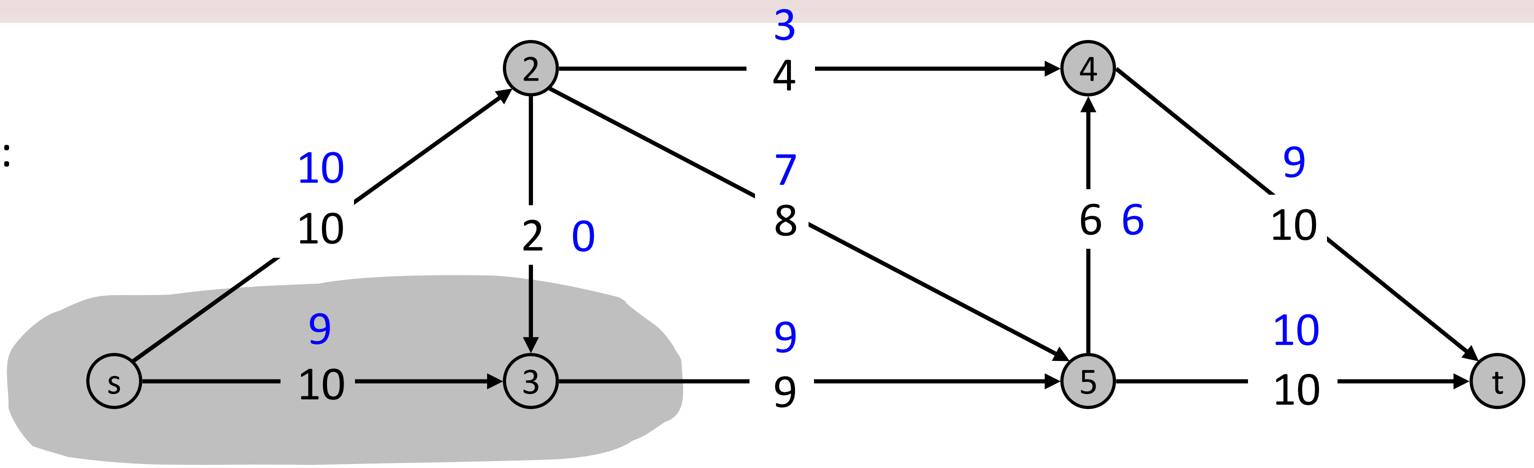
Flow value = 19

G_f:



Ford-Fulkerson Algorithm

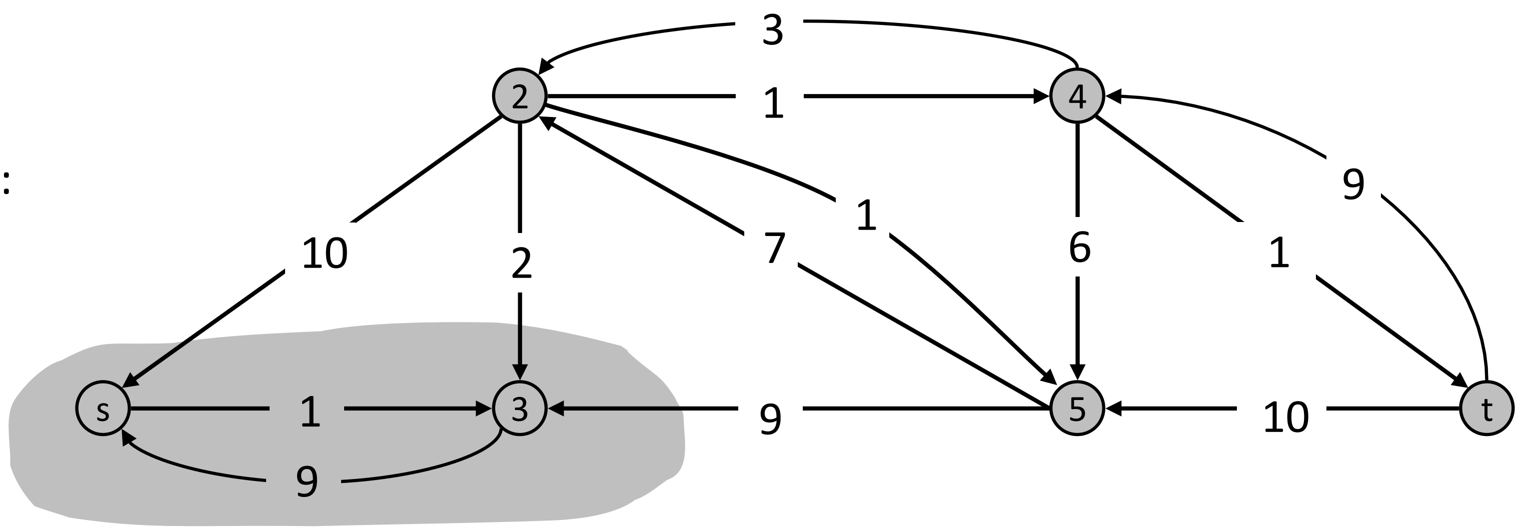
G:



Cut capacity = 19

Flow value = 19

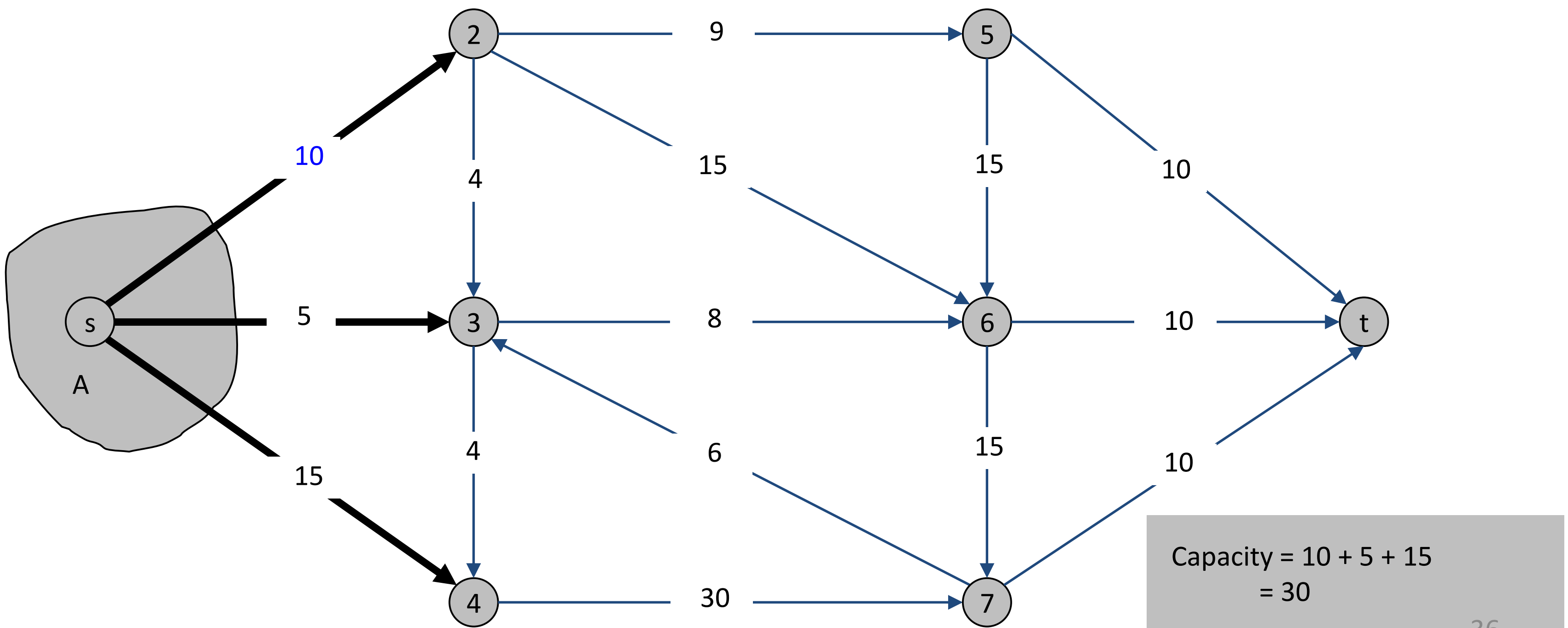
G_f:



Cuts

Def. An s-t cut is a partition (A, B) of V with $s \in A$ and $t \in B$.

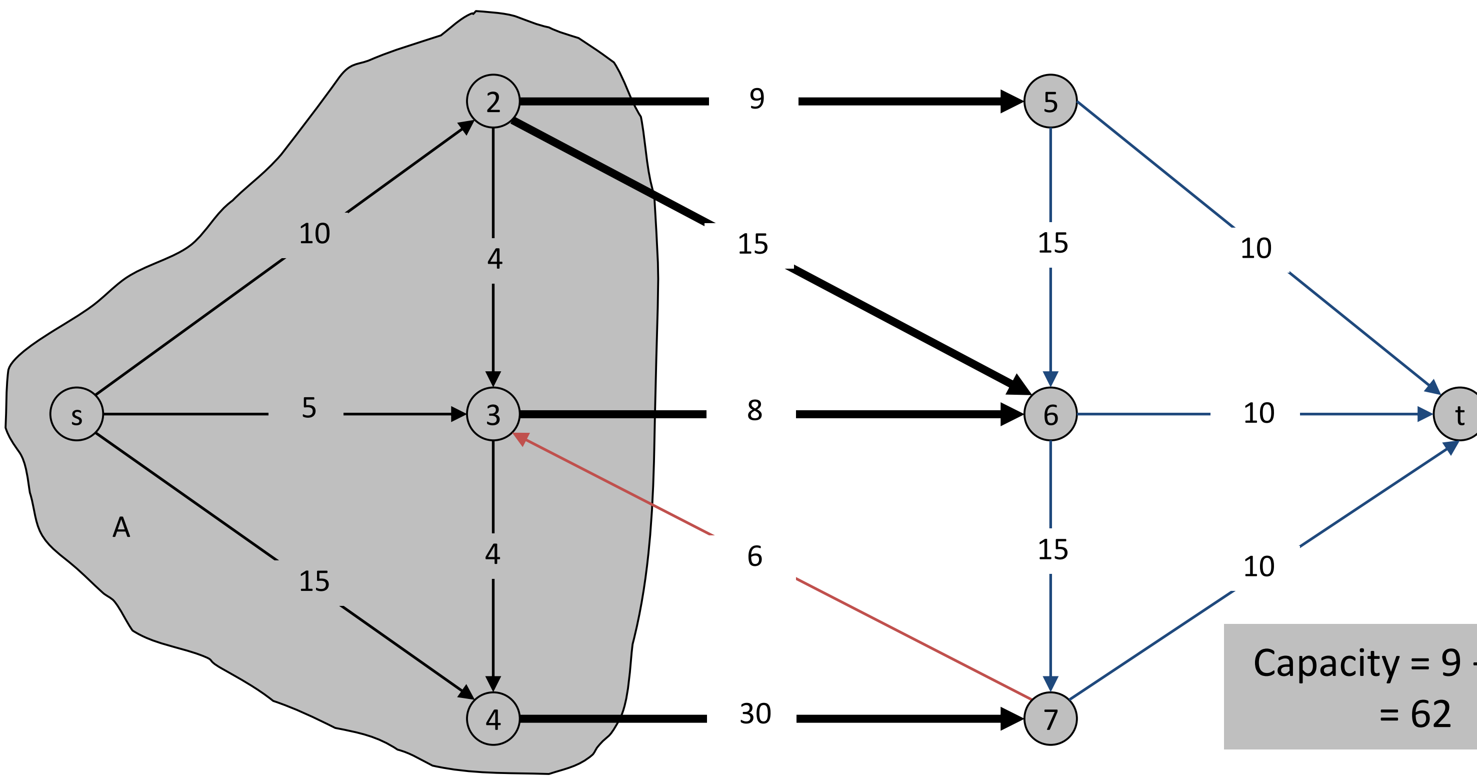
Def. The capacity of a cut (A, B) is: $$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



Cuts

Def. An *s-t cut* is a partition (A, B) of V with $s \in A$ and $t \in B$.

Def. The capacity of a cut (A, B) is: $$cap(A,B) = \sum_{e \text{ out of } A} c(e)$$



Capacity = 9 + 15 + 8 + 30
= 62

Minimum Cut Problem

Min s-t cut problem. Find an s-t cut of minimum capacity.