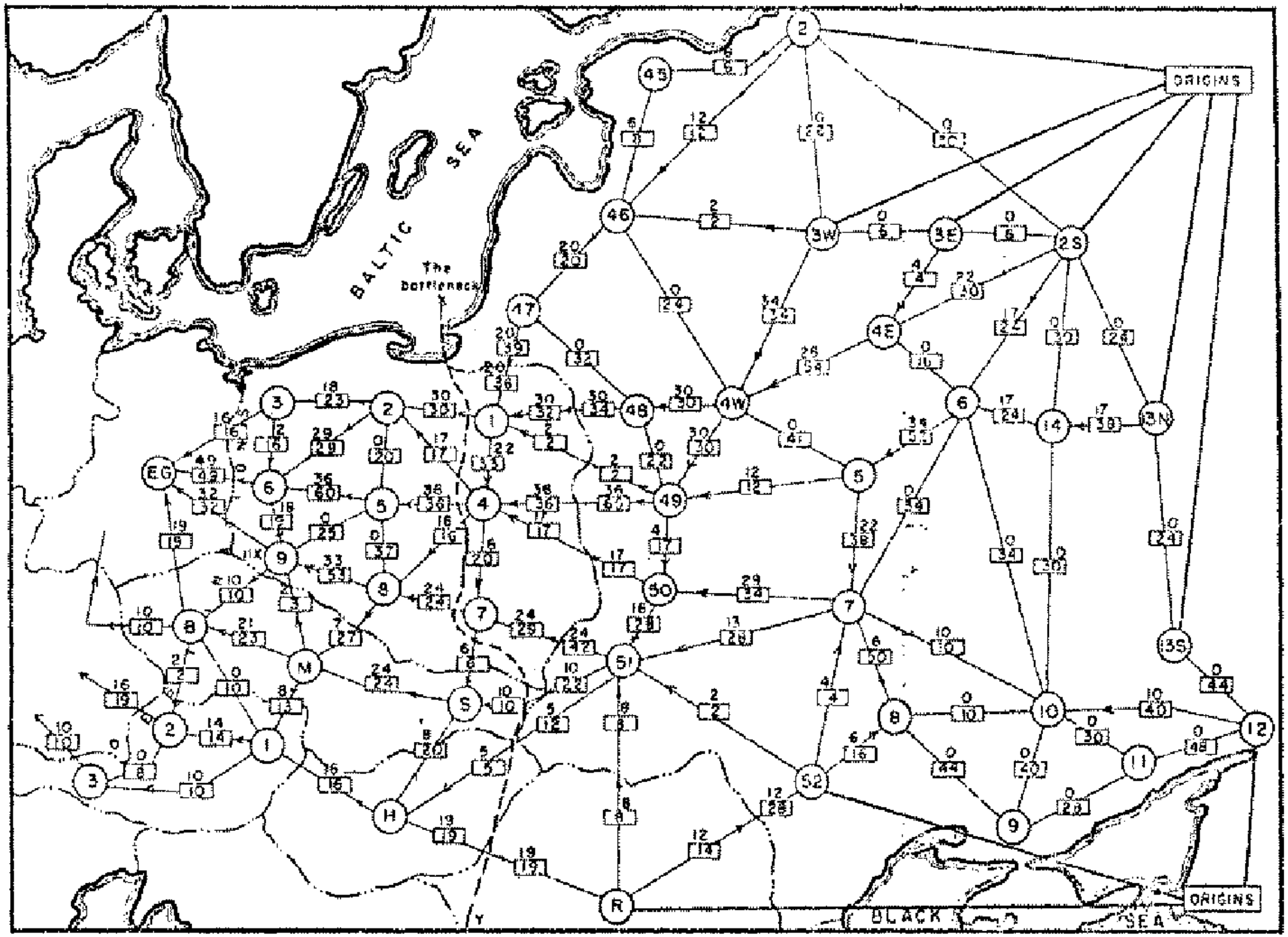
Maximum Flow

Soviet Rail Network, 1955



Reference: *On the history of the transportation and maximum flow problems.*
Alexander Schrijver in Math Programming, 91: 3, 2002.

Maximum Flow and Minimum Cut

Max flow and min cut.

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

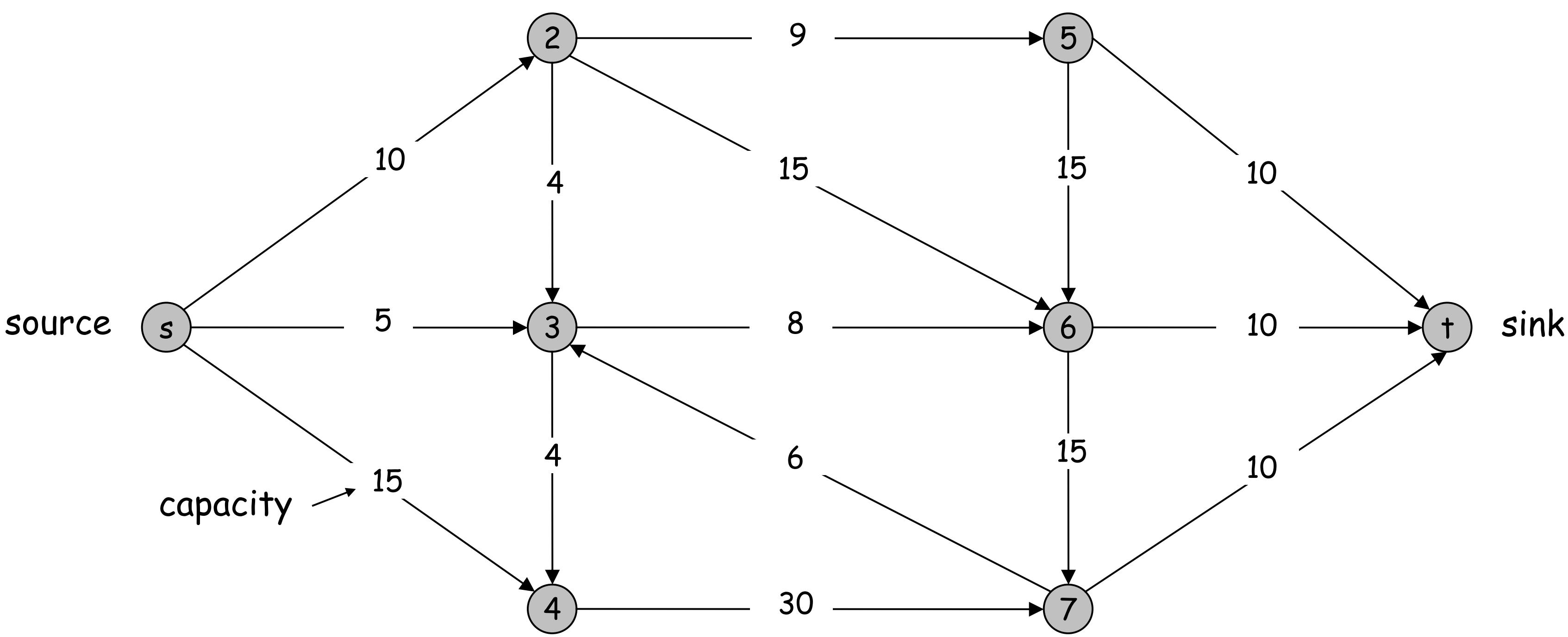
Nontrivial applications / reductions.

- 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.
- Many many more . . .

Minimum Cut Problem

Flow network.

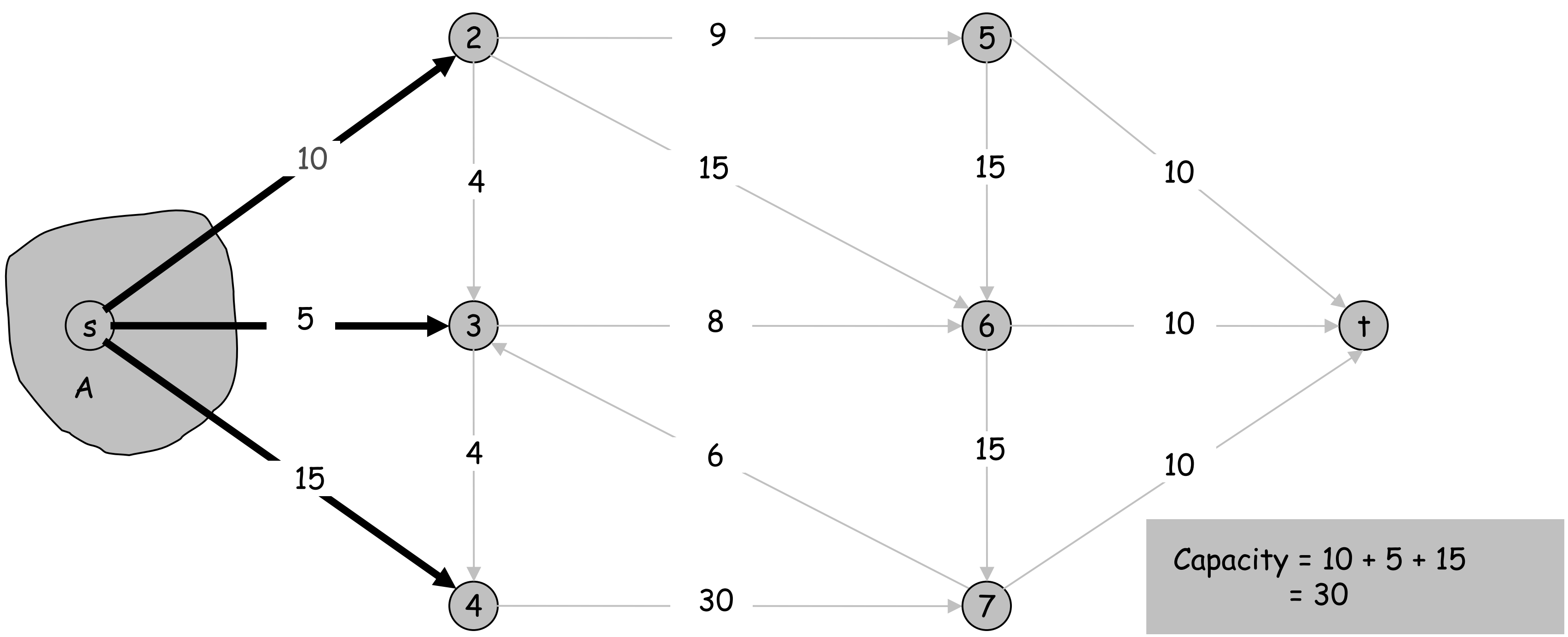
- Abstraction for material flowing through the edges.
- G = (V, E) = directed graph, no parallel edges.
- Two distinguished nodes: s = source, t = sink.
- c(e) = capacity of edge e.



Cuts

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

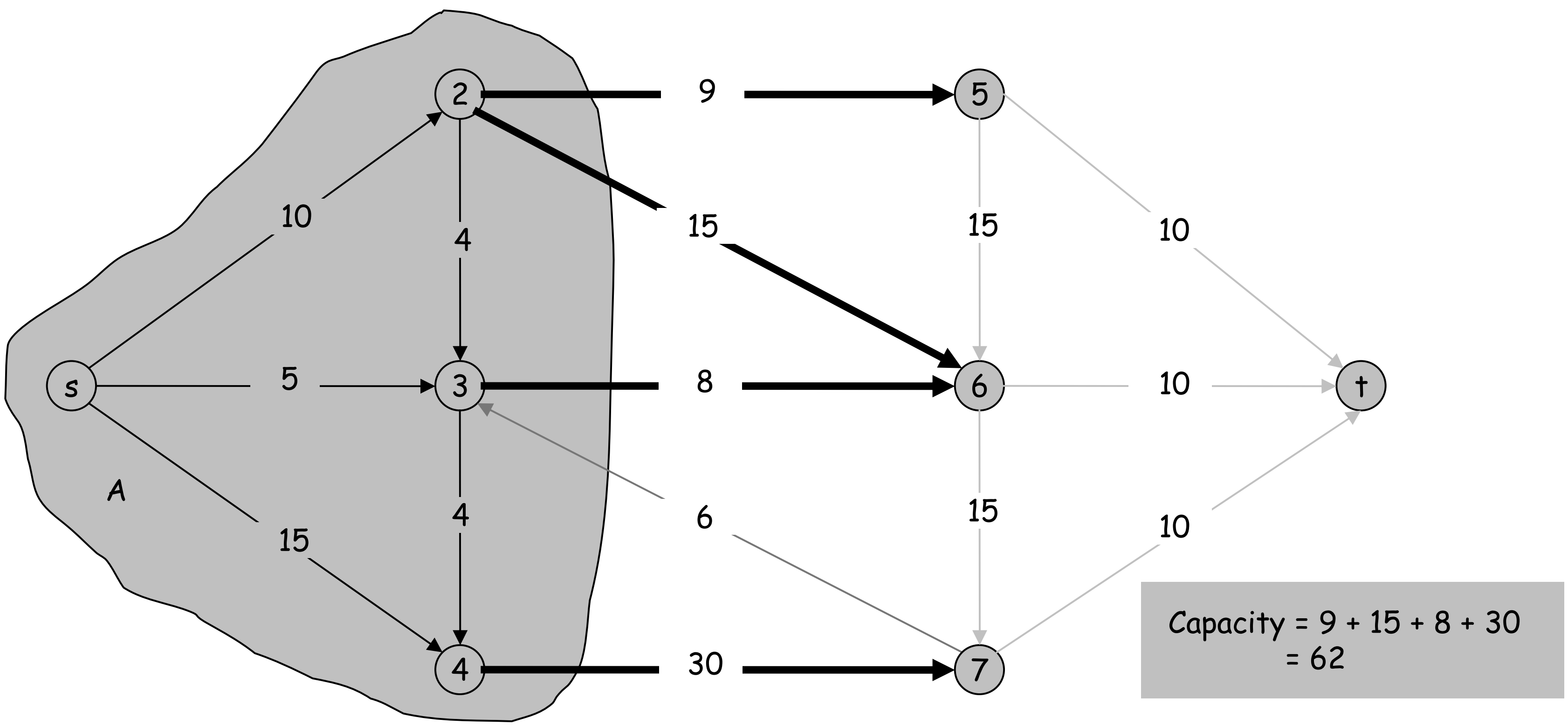
Def. The capacity of a cut (A, B) is: $$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$



Cuts

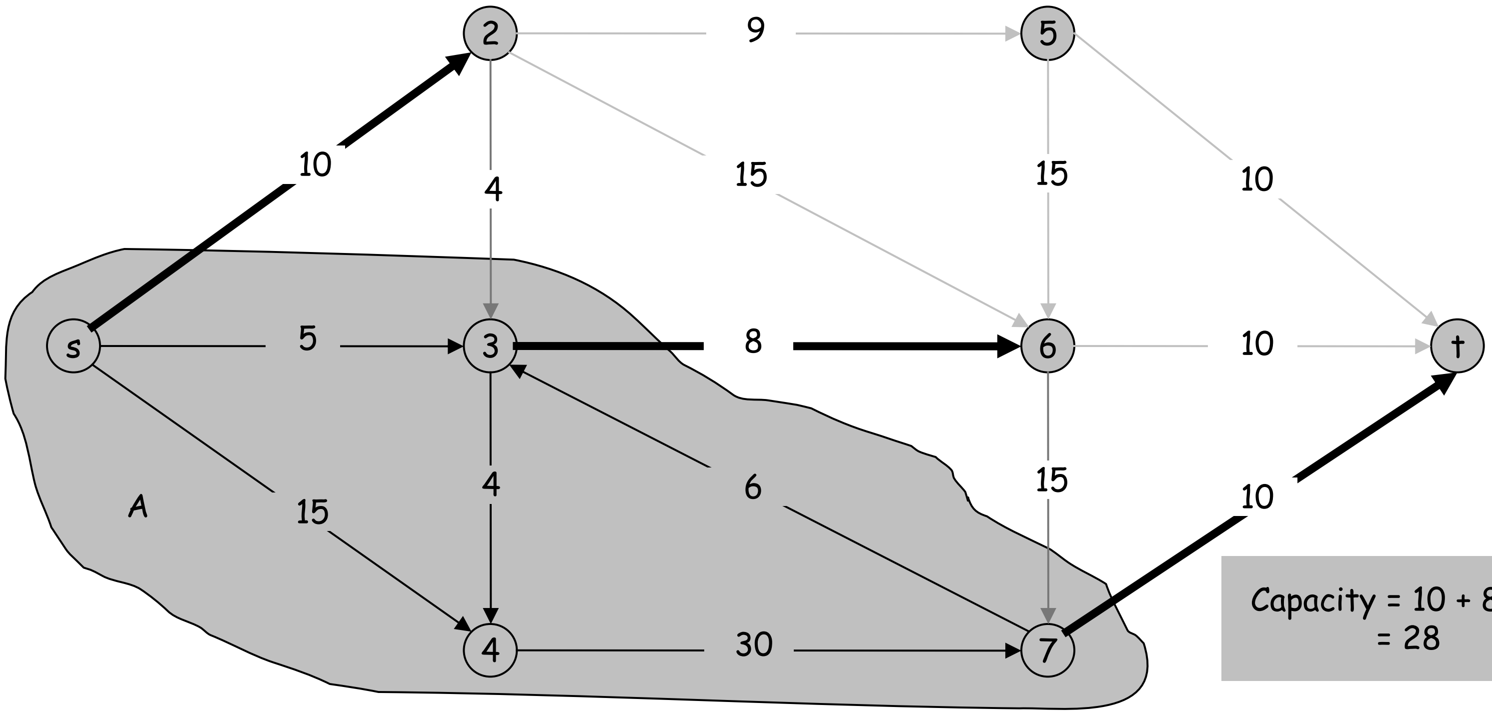
Def. An s-t cut is a partition (A, B) of V with s $\in$ A and t $\in$ B.

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



Minimum Cut Problem

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



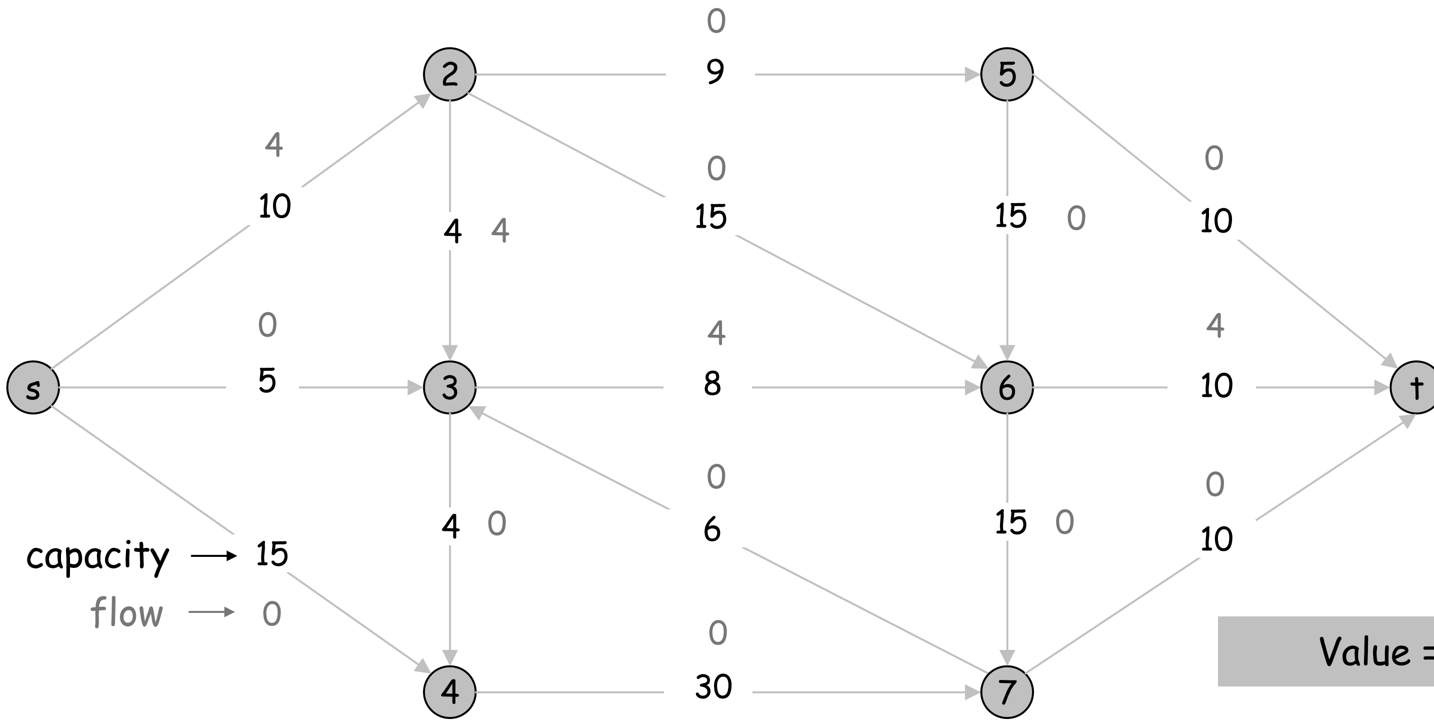
Capacity = 10 + 8 + 10
= 28

Flows

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

- For each $e \in E$: $0 \le f(e) \le c(e)$ (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: $v(f) = \sum_{e \text{ out of } s} f(e)$.



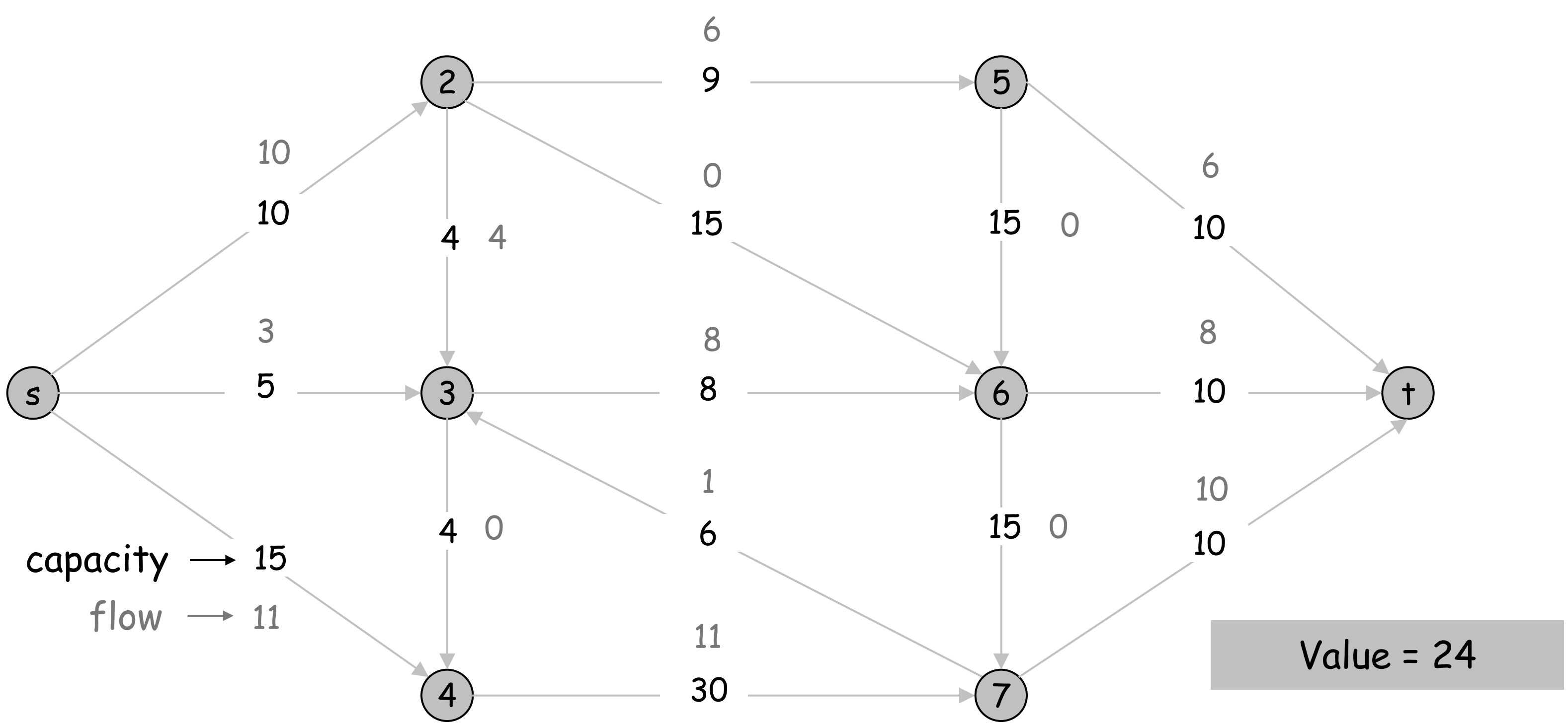
Value = 4

Flows

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

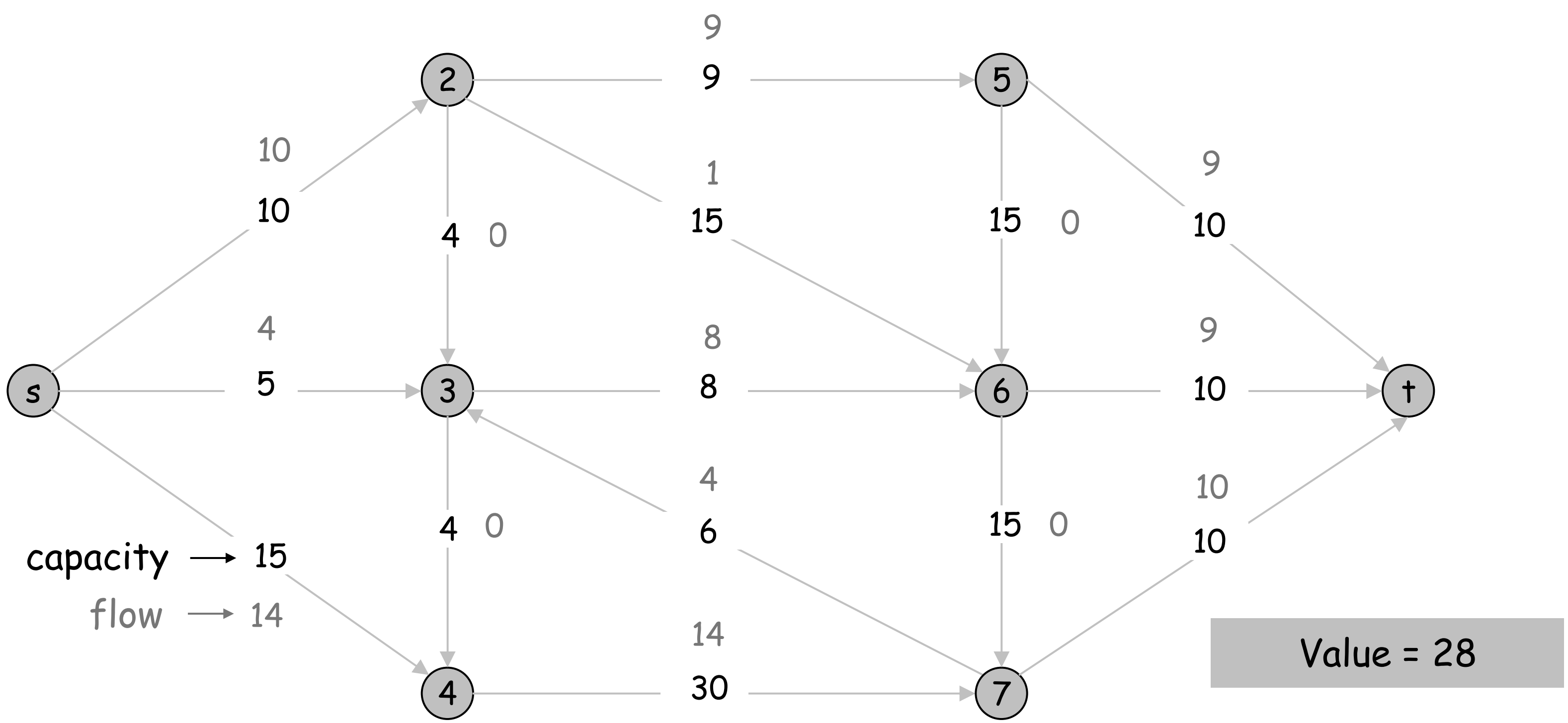
- For each $e \in E$: $0 \le f(e) \le c(e)$ (capacity)
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ (conservation)

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



Maximum Flow Problem

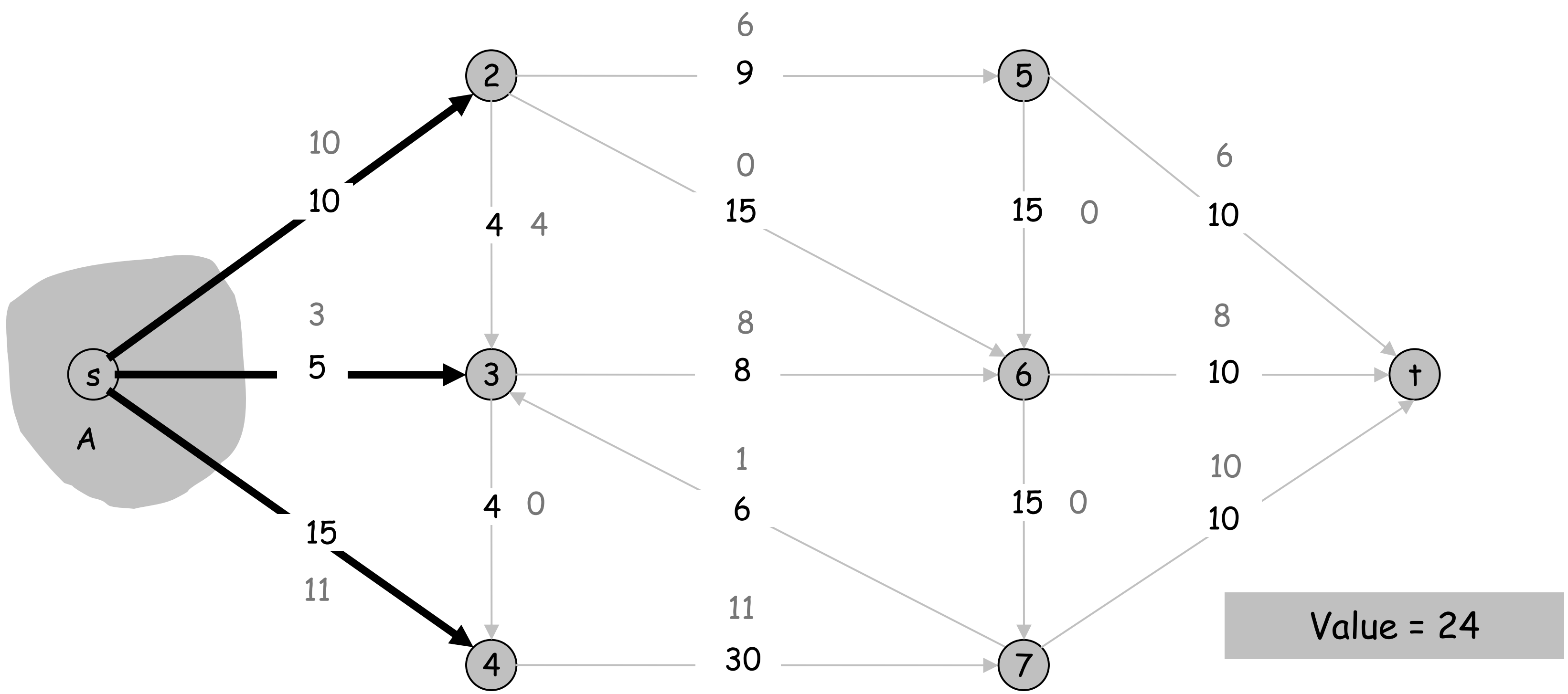
Max flow problem. Find s-t flow of maximum value.



Flows and Cuts

Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

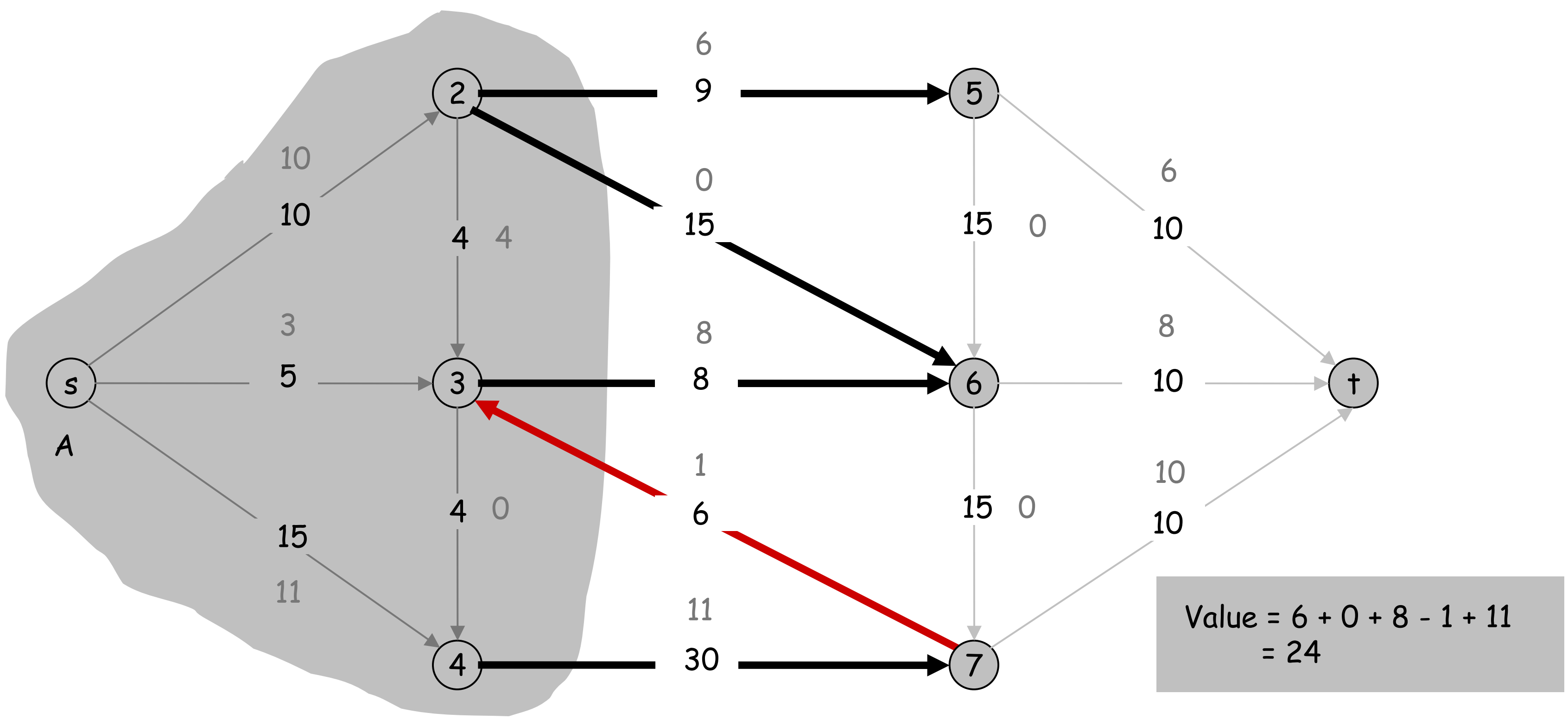
$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$



Flows and Cuts

Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

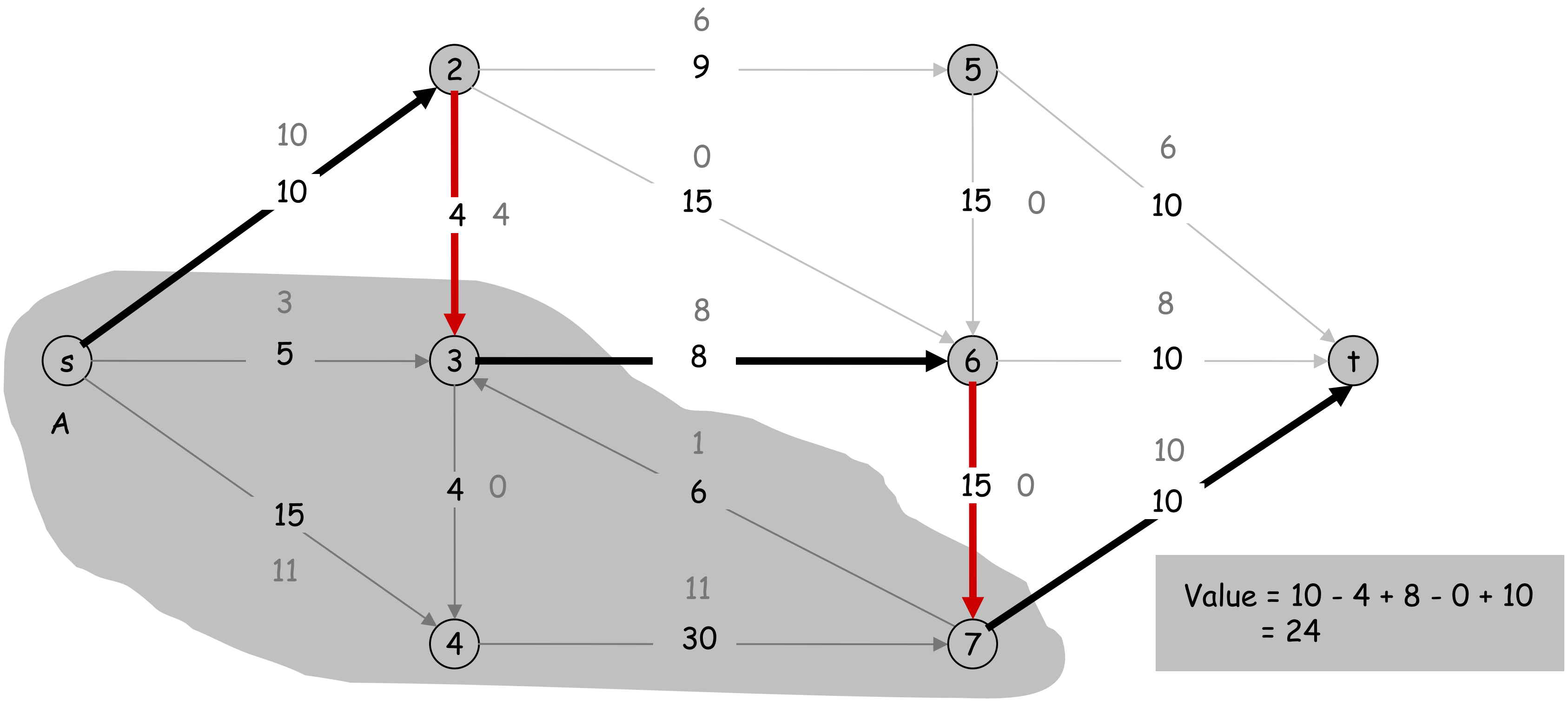
$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$



Flows and Cuts

Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving s.

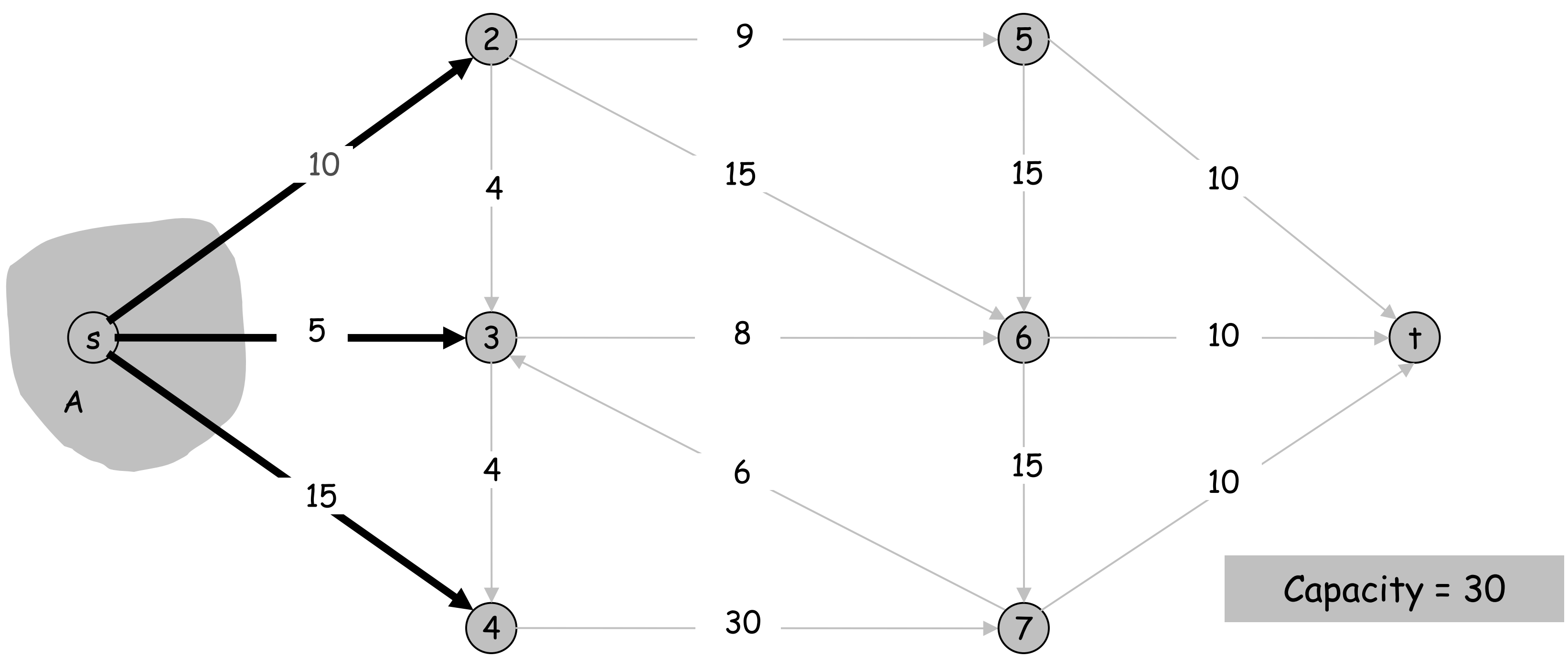
$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \;=\; v(f)$$



Flows and Cuts

Weak duality. Let f be any flow, and let (A, B) be any s-t cut. Then the value of the flow is at most the capacity of the cut.

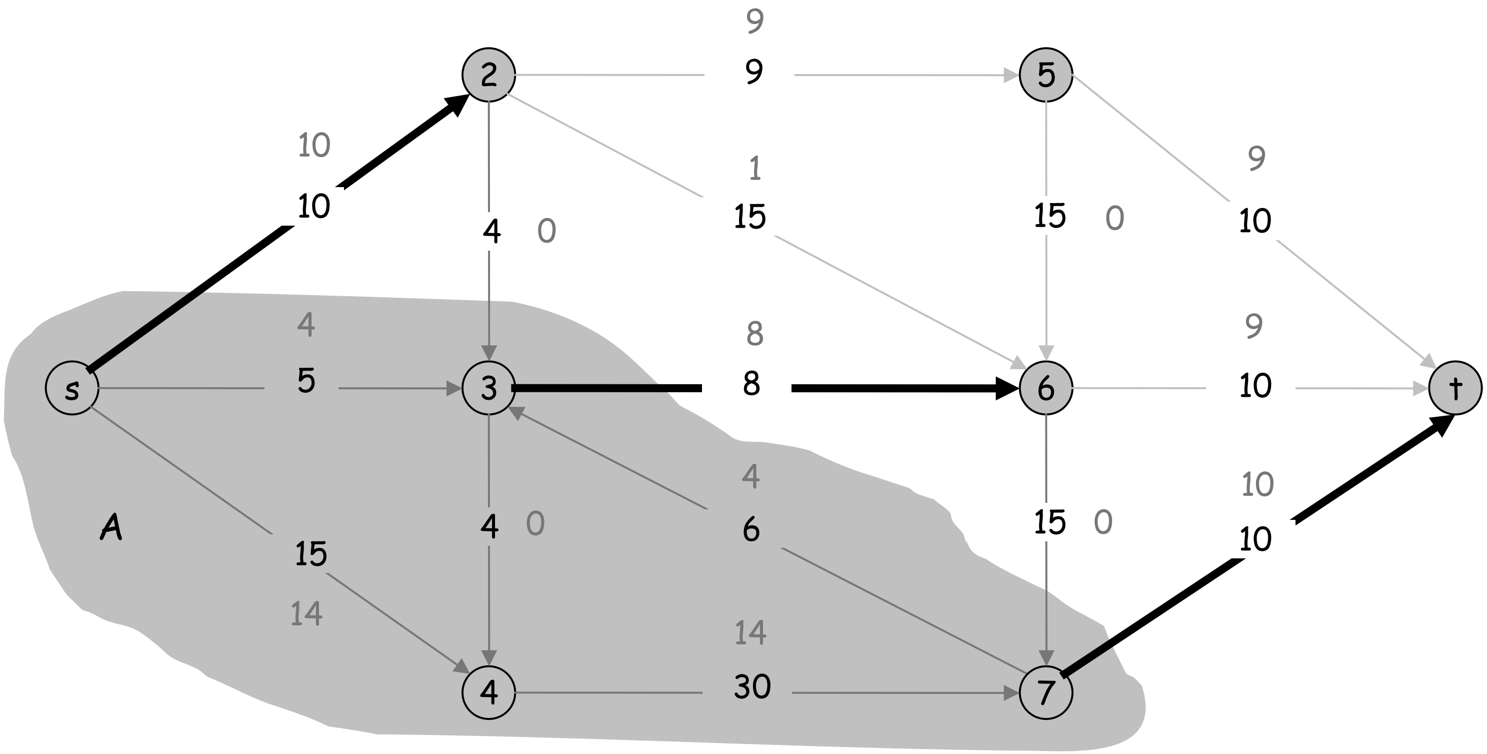
Cut capacity = 30 $\Rightarrow$ Flow value $\leq$ 30



Certificate of Optimality

Corollary. Let f be any flow, and let (A, B) be any cut.
If v(f) = cap(A, B), then f is a max flow and (A, B) is a min cut.

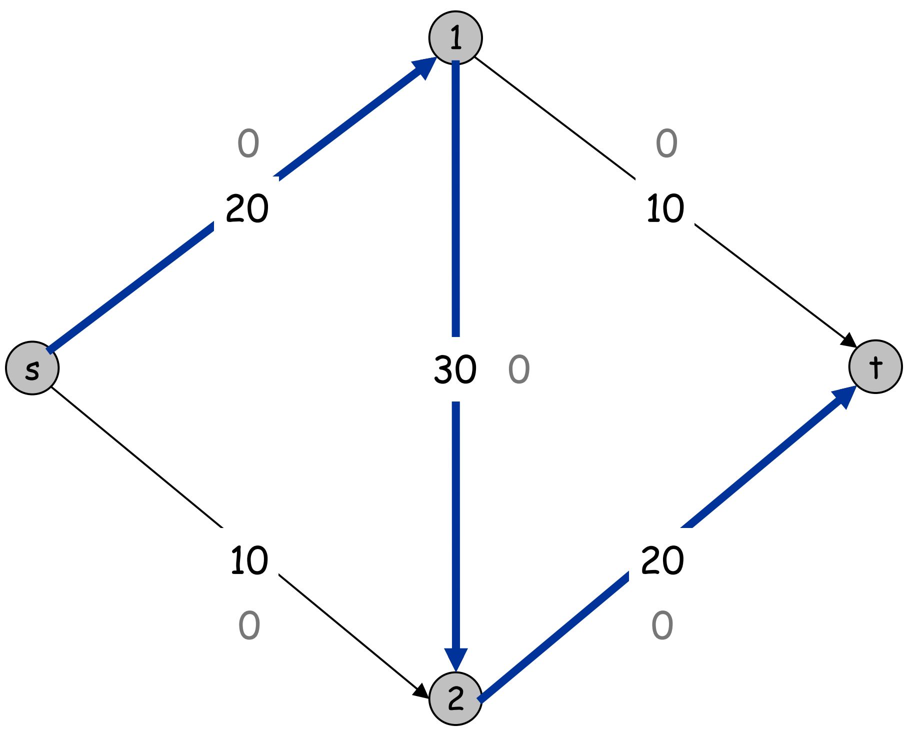
Value of flow = 28
Cut capacity = 28 $\Rightarrow$ Flow value $\leq$ 28



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.

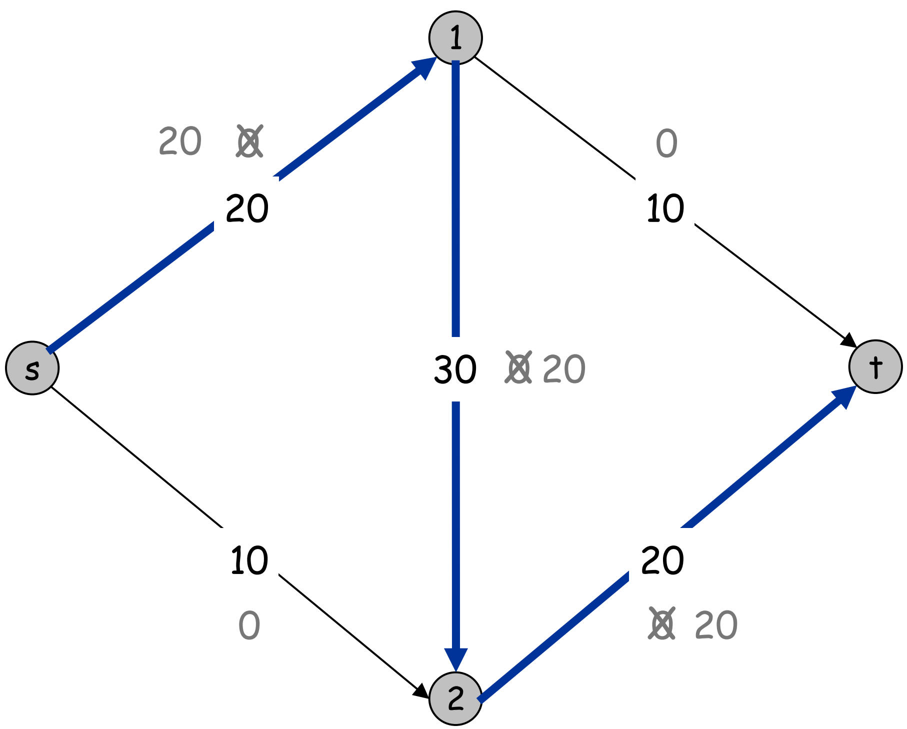


Flow value = 0

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.



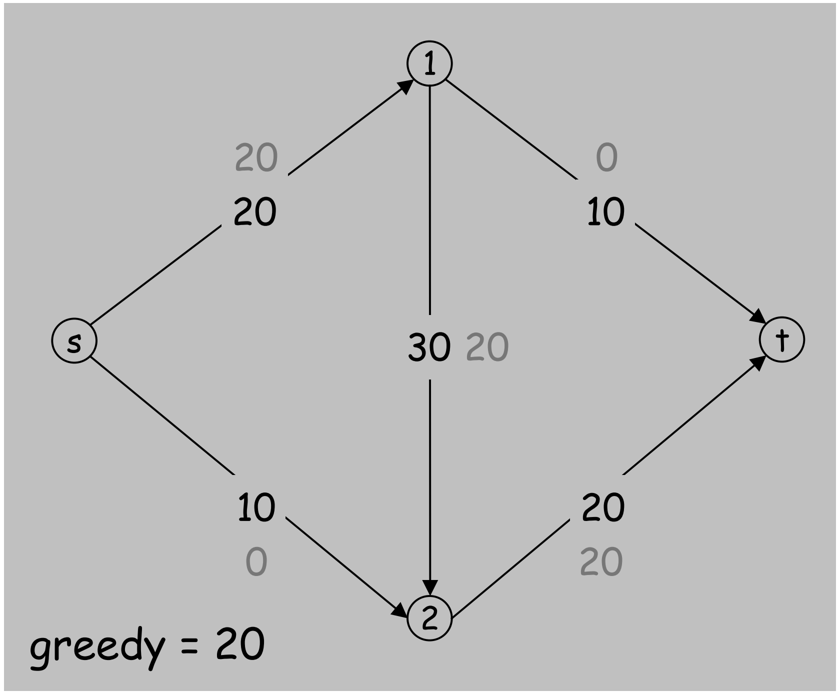
Flow value = 20

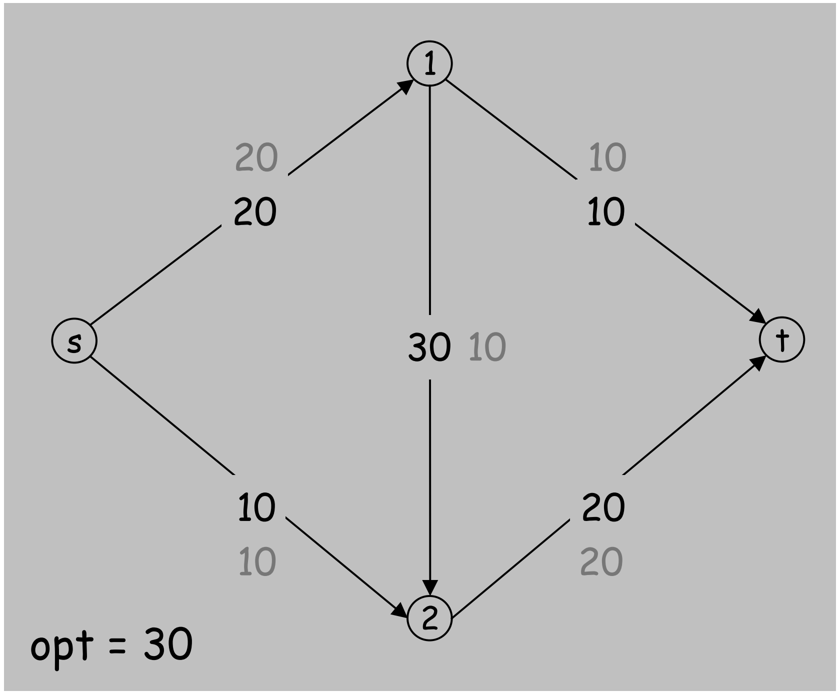
Towards a Max Flow Algorithm

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





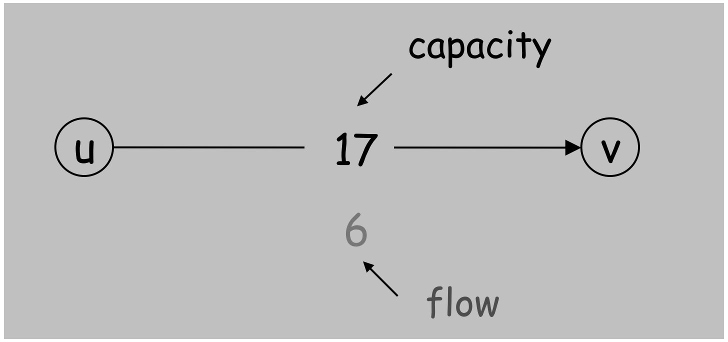
Ford-Fulkerson Method

Has different implementations with different running times

Based on 3 main ideas:

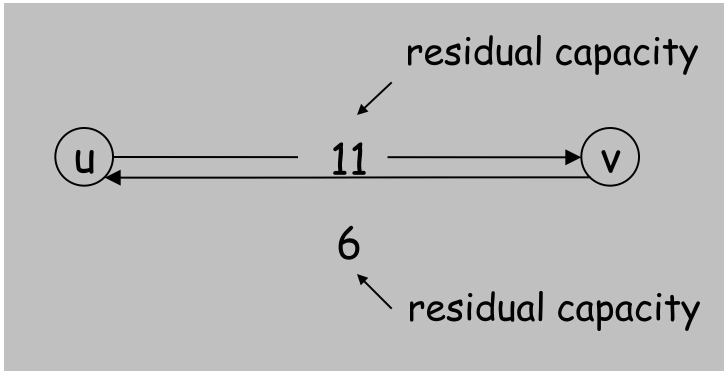
- Residual graphs
- Augmenting paths
- Cuts

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:

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

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

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

Augmenting Path Algorithm

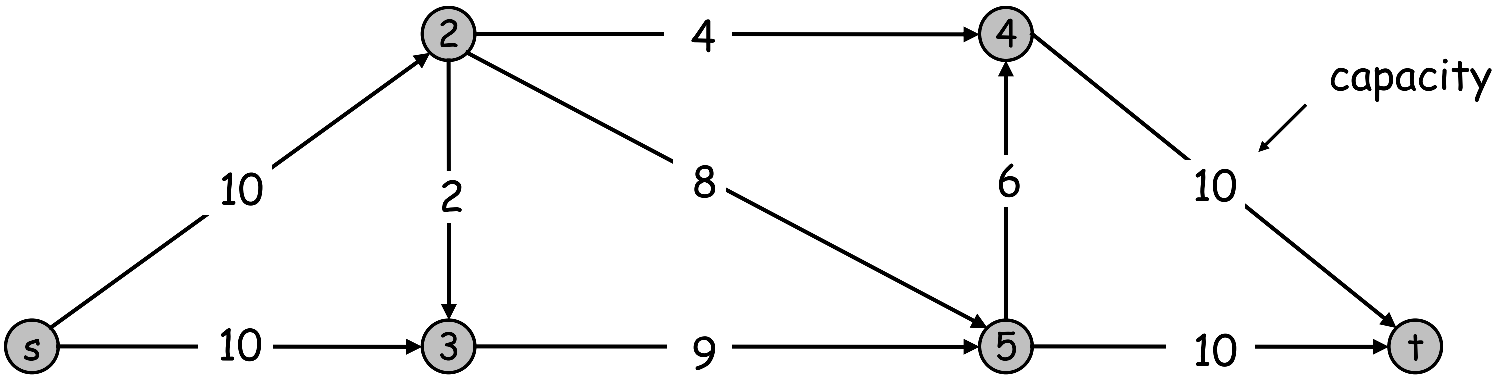
```
Ford-Fulkerson(G, s, t, c) {
   for each edge (u,v) ∈ E
       (u,v).f ← 0
   Gf ← residual graph

   while (exists an augumentative path P from s to t in Gf)
   {
      cf(p) = min {cf(u,v) : (u,v) is in P} // bottleneck
      for each (u,v) in P {
      if ((u,v) ∈ E)
    (u,v).f ← (u,v).f + cf(p)                   // forward edge
      else      (v,u).f ← (v,u).f - cf(p)       // reverse edge

      update Gf
   }
   return f
}
```

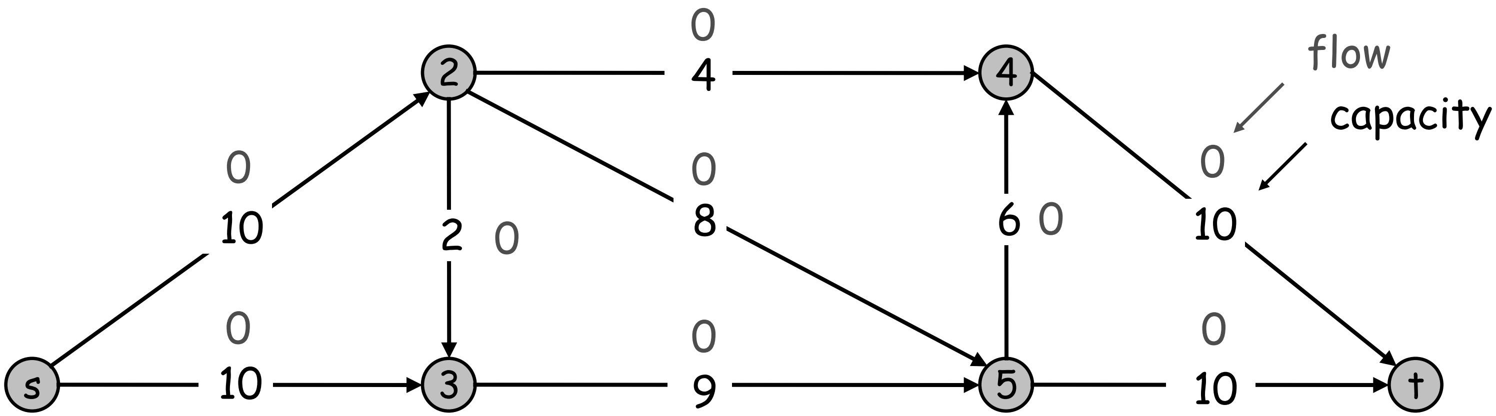
Ford-Fulkerson Algorithm

G:



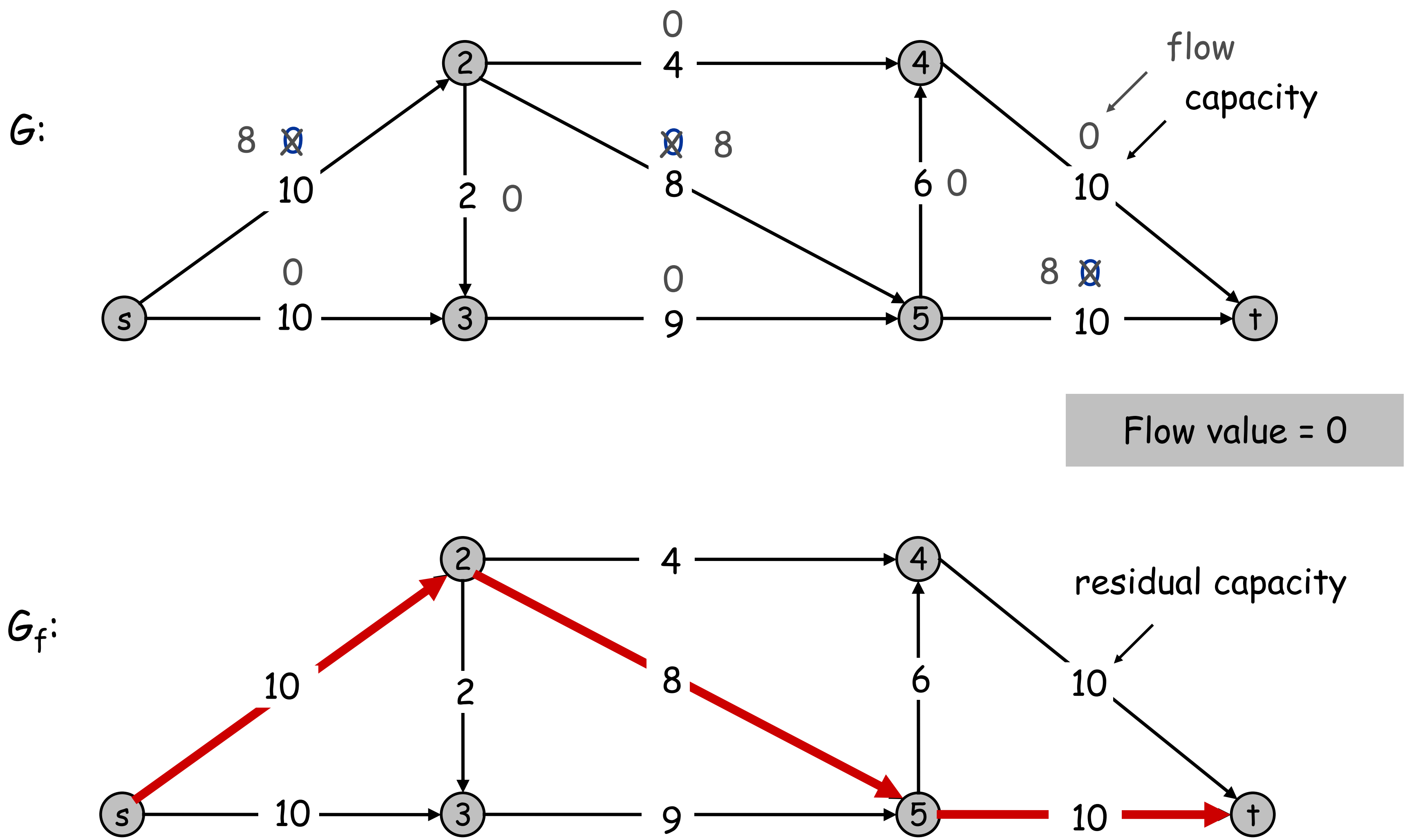
Ford-Fulkerson Algorithm

G:



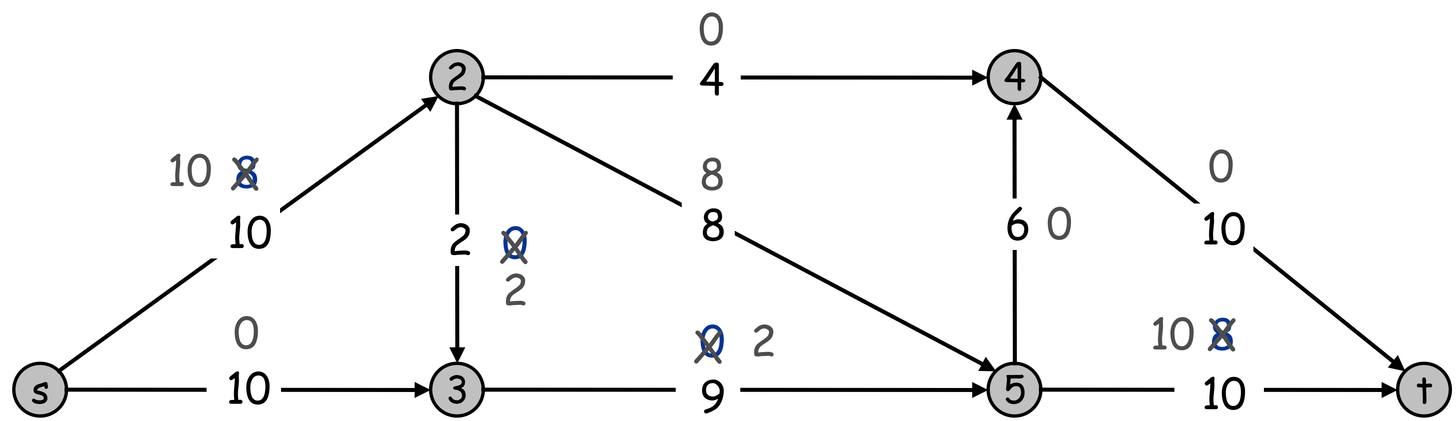
Flow value = 0

Ford-Fulkerson Algorithm



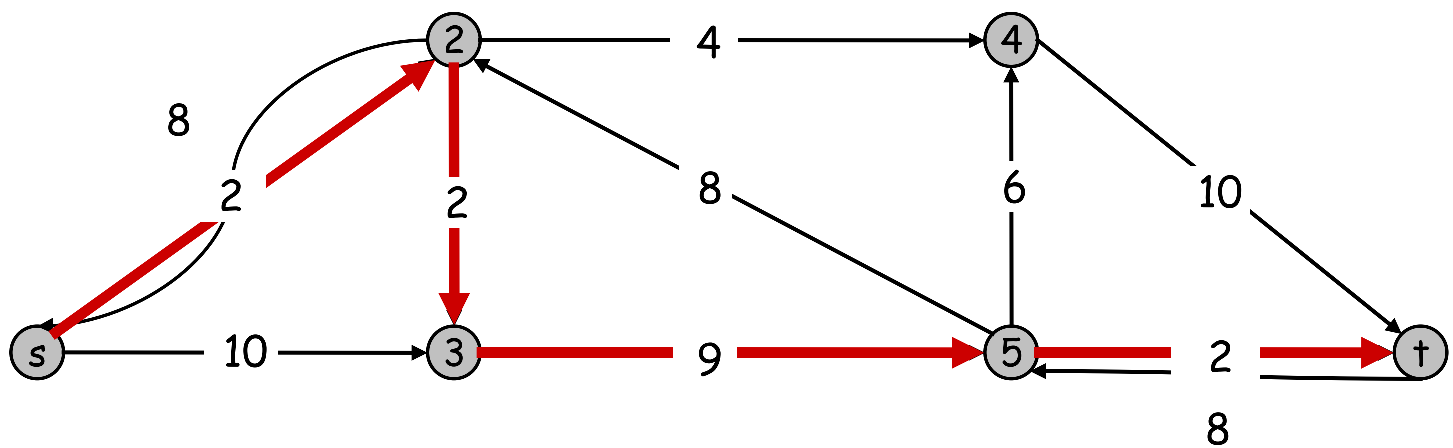
Ford-Fulkerson Algorithm

G:



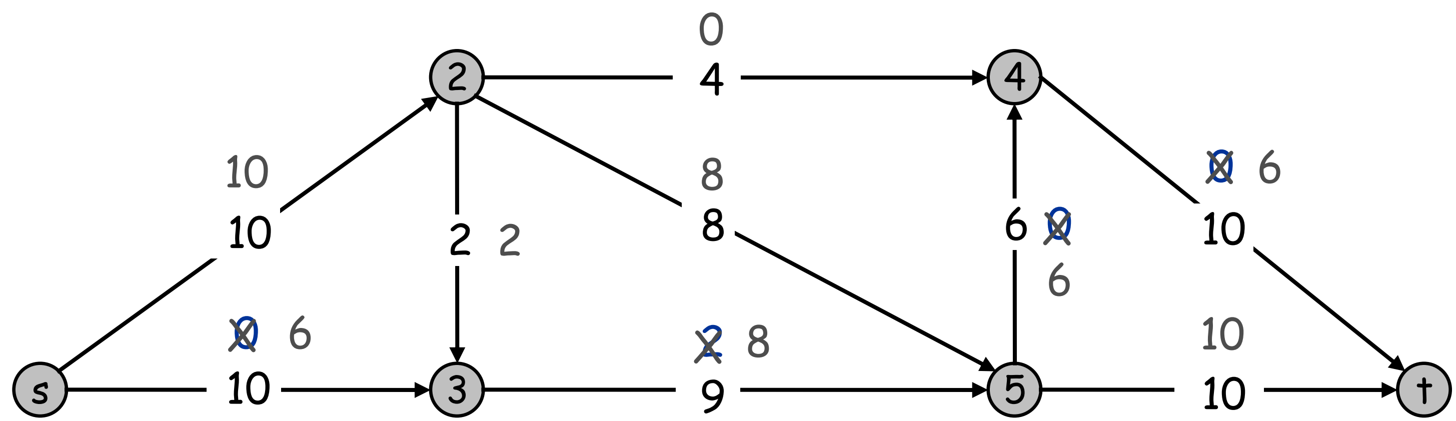
Flow value = 8

G_f:



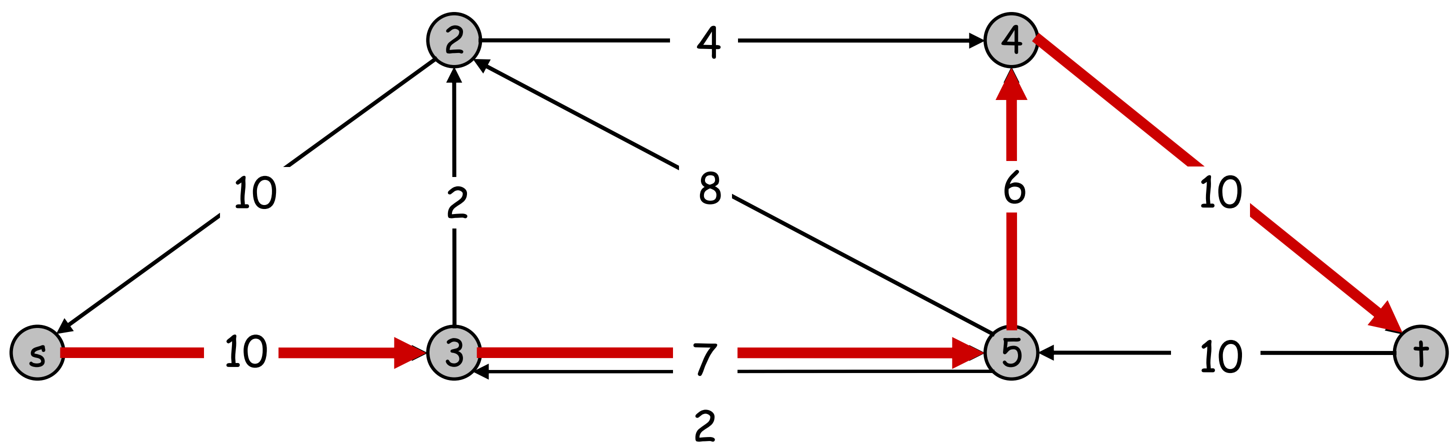
Ford-Fulkerson Algorithm

G:



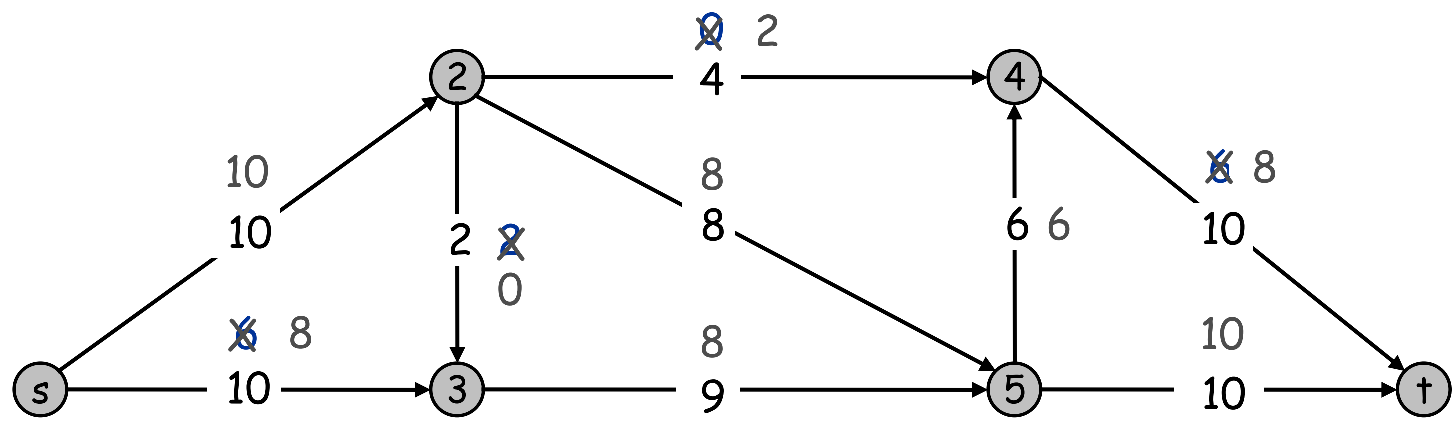
Flow value = 10

G_f:



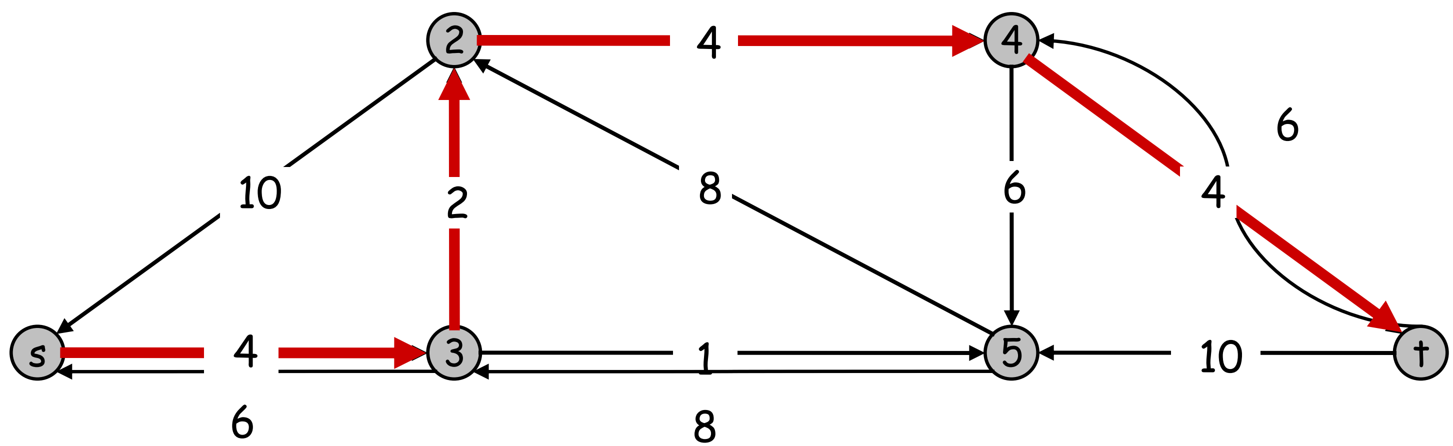
Ford-Fulkerson Algorithm

G:

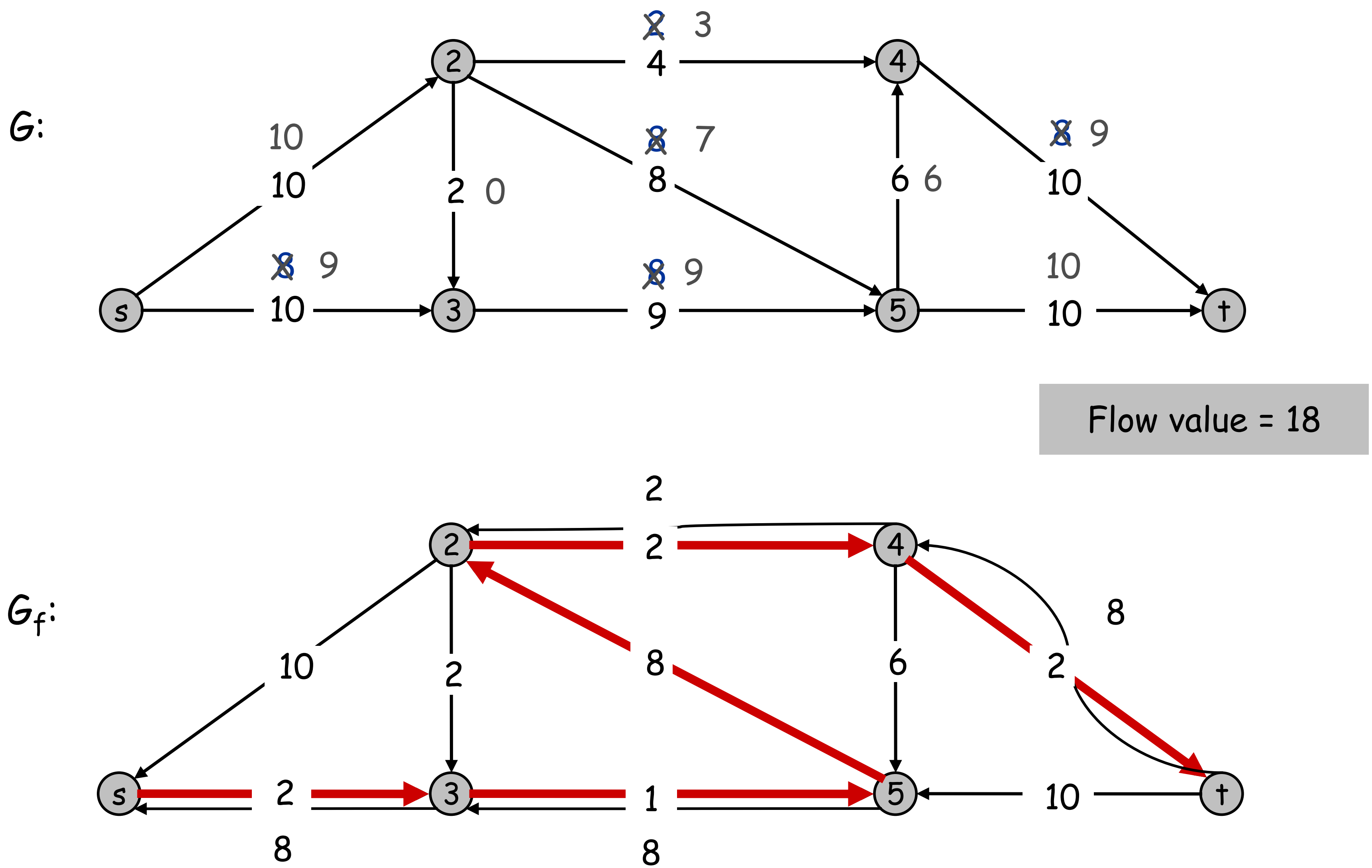


Flow value = 16

G_f:

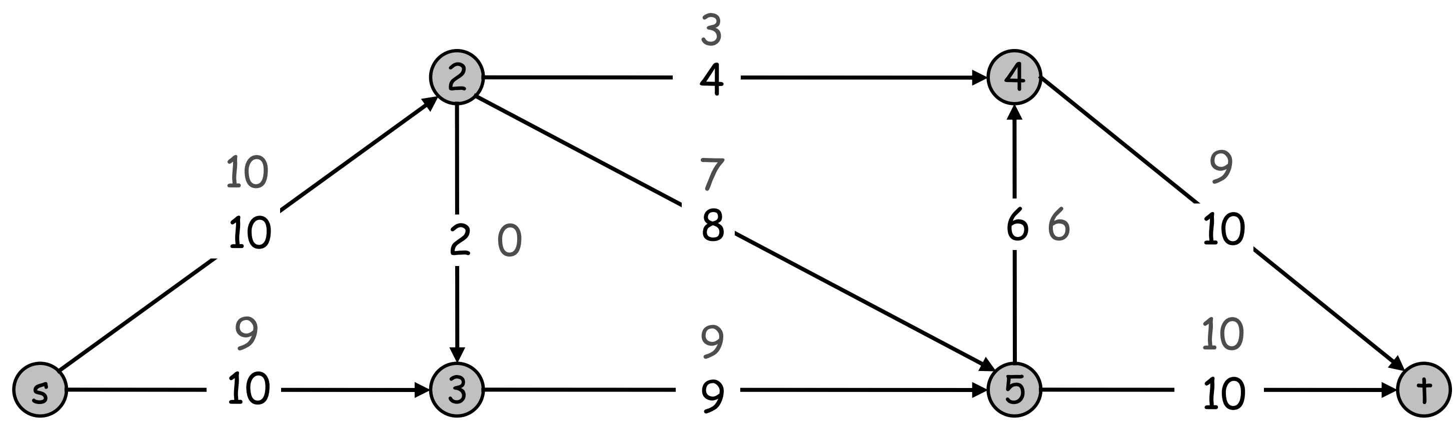


Ford-Fulkerson Algorithm



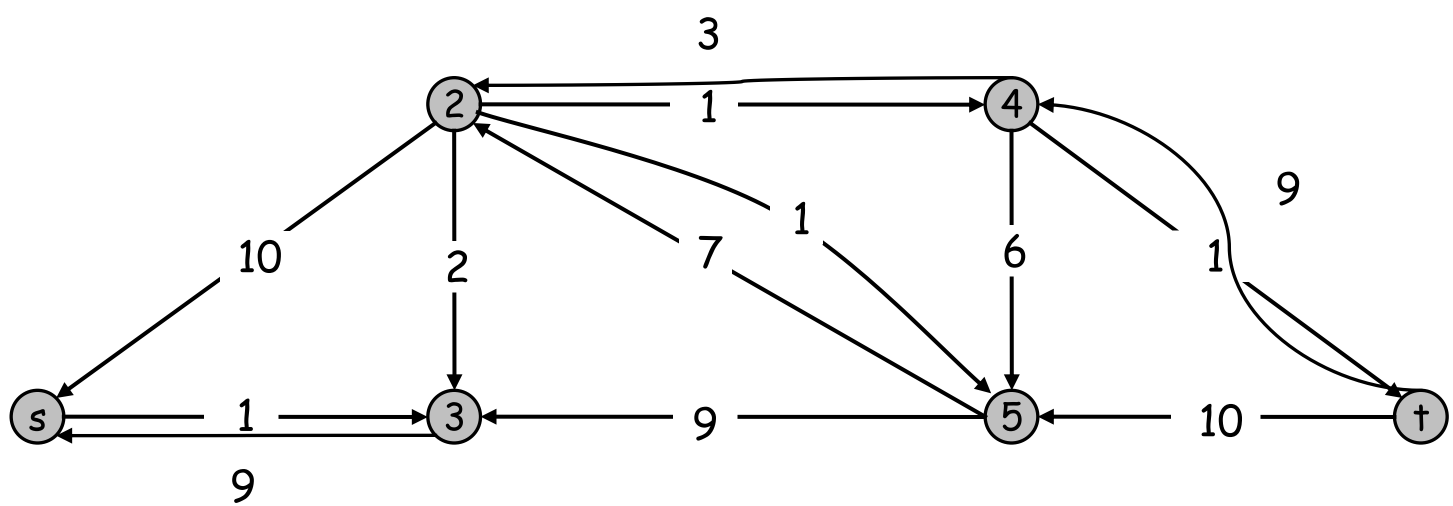
Ford-Fulkerson Algorithm

G:

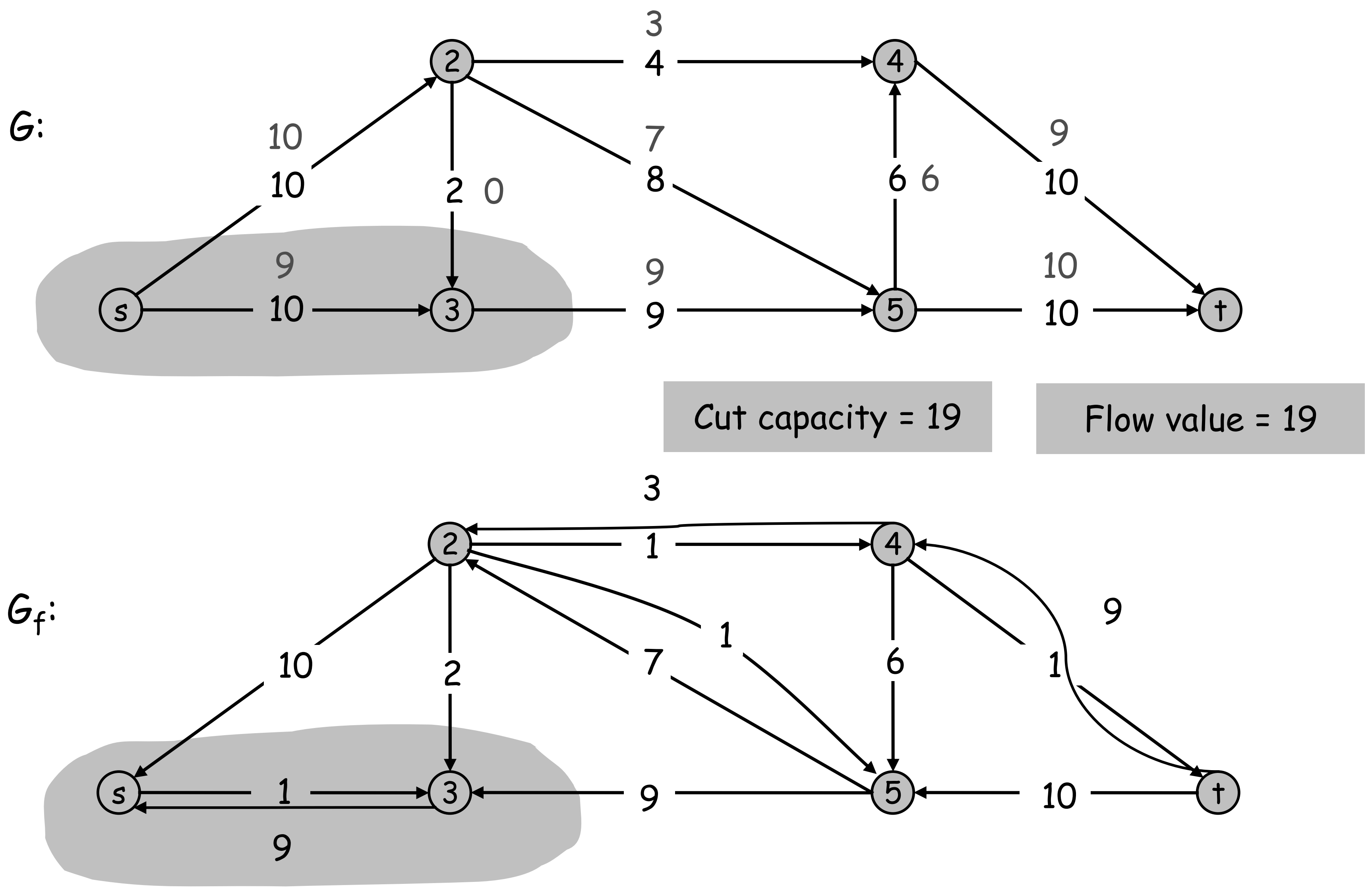


Flow value = 19

G_f:



Ford-Fulkerson Algorithm



Augmenting Path Algorithm - Run time analysis?

```
     Ford-Fulkerson(G, s, t, c) {
|E|     for each edge (u,v) ∈ E
            (u,v).f ← 0
        G_f ← residual graph

|f|     while (exists an augmentative path P from s to t in G_f)
        {
           c_f(p) = min {c_f(u,v) : (u,v) is in P} // bottleneck
           for each (u,v) in P {
           if ((u,v) ∈ E)
          (u,v).f ← (u,v).f + c_f(p)             // forward edge
           else       (v,u).f ← (v,u).f - c_f(p)  // reverse edge

           Update G_f      |E|
        }
        return f
     }
```

O(E |f|), considering a polynomial algorithm is used to find the paths

7.3 Choosing Good Augmenting Paths

Choosing Good Augmenting Paths

Use care when selecting augmenting paths.

- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.
- If capacities are irrational, algorithm not guaranteed to terminate!

Goal: choose augmenting paths so that:

- Can find augmenting paths efficiently.
- Few iterations.

Choose augmenting paths with: [Edmonds-Karp 1972, Dinitz 1970]

- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.

Edmonds-Karp

Improves Ford-Fulkerson

- Uses a breadth-first search to find augmentative paths
- Chooses the path with the **smallest number of egdes**

```
Edmonds-Karp(G, s, t, c) {
   for each edge (u,v) ∈ E
       (u,v).f ← 0
   Gf ← residual graph
   P ← find smallest augmentative path s-t in Gf using BFS
   while (P exists)
   {
      cf(p) = min {cf(u,v) : (u,v) is in P} // bottleneck
      for each (u,v) in P {
      if ((u,v) ∈ E)
    (u,v).f ← (u,v).f + cf(p)
      else     (v,u).f ← (v,u).f - cf(p)

      Update Gf |E|
      P ← find smallest augmentative path s-t in Gf using
BFS
   }
   return f
}
```

Complexity Analysis

Runtime = |E| x number of augmentative paths

Number of augmentative paths

- An edge (u,v) in a path P in a residual graph G_f is critical if the residual capacity of (u,v) equals the residual capacity of P
- After augmenting P, critical edges disappear from G_f
- If u and v are vertices connected by an edge in E
- Since augmentative paths are shortest paths, when (u,v) is critical

$$\delta_f(s, v) = \delta_f(s, u) + 1$$

- When the flow is updated, (u,v) disappears from G_f
- It cannot reappear unless flow from u to v is decreased
  - Only happens if (v,u) appears in an augmentative path
  - In this case, $\delta_{f'}(s, u) = \delta_f(s, u) + 2$

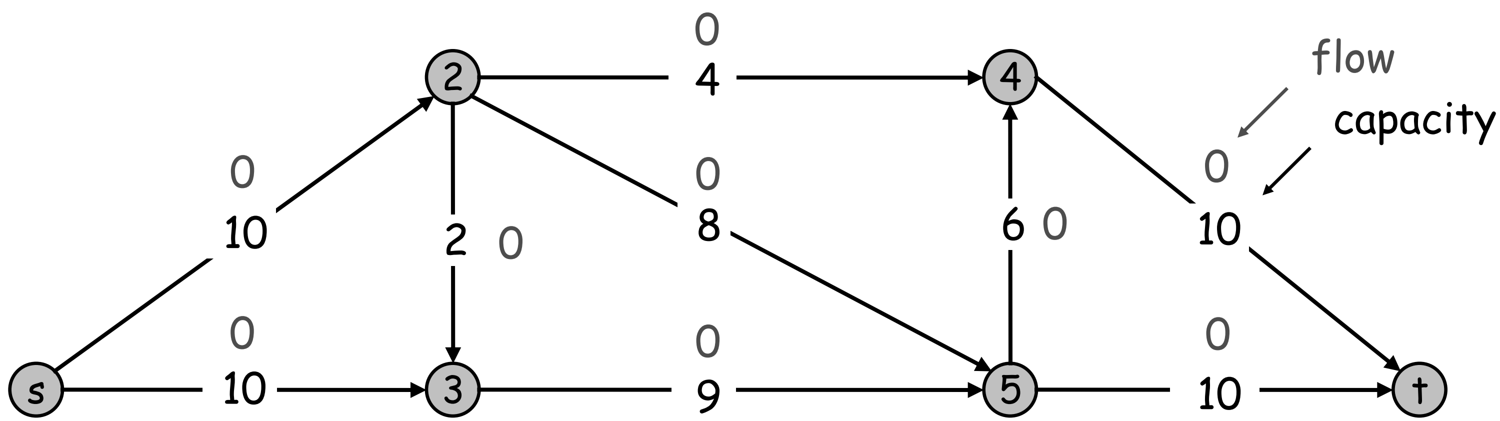
Complexity Analysis

- From the time (u,v) becomes critical until the next time it becomes critical, the distance of s to u increases by at least 2
- The intermediate nodes in path (s,u) cannot contain s, u or t
  - To u become unreachable from s, its distance has to be at most |V-2|
- After the first time u becomes critical, it can become critical at most (|v-2|)/2 times
- As there are |E| pairs of vertices, we can have |E|x|V| critical vertices during the execution of the algorithm

- Runtime = |E| x number of augmentative paths
  = |E| x |VE| = $O(VE^2)$

Edmonds-Karp Algorithm

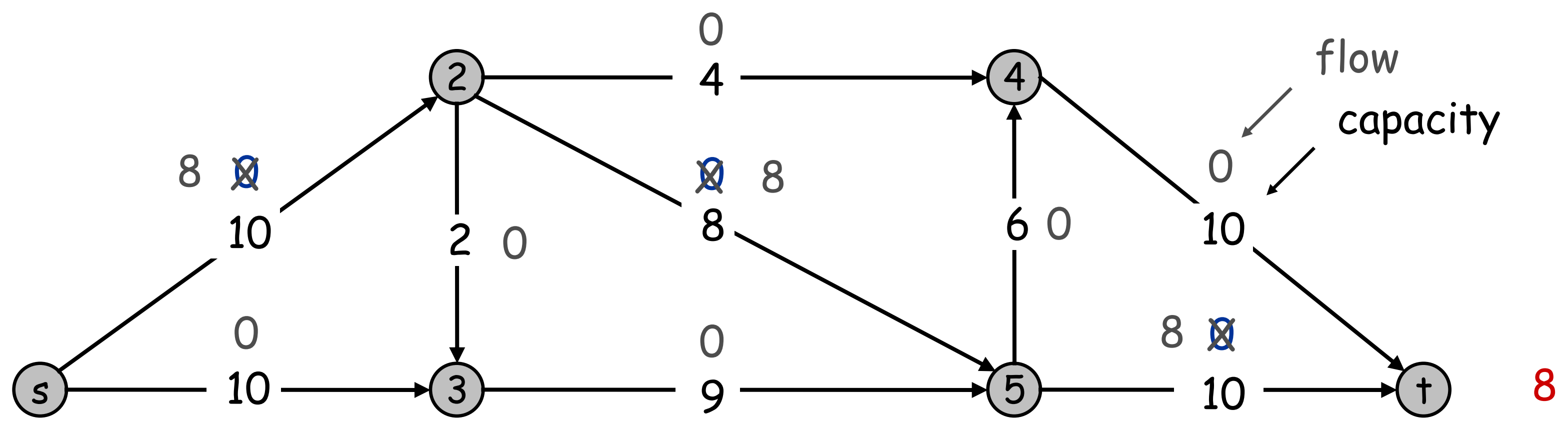
G:



Flow value = 0

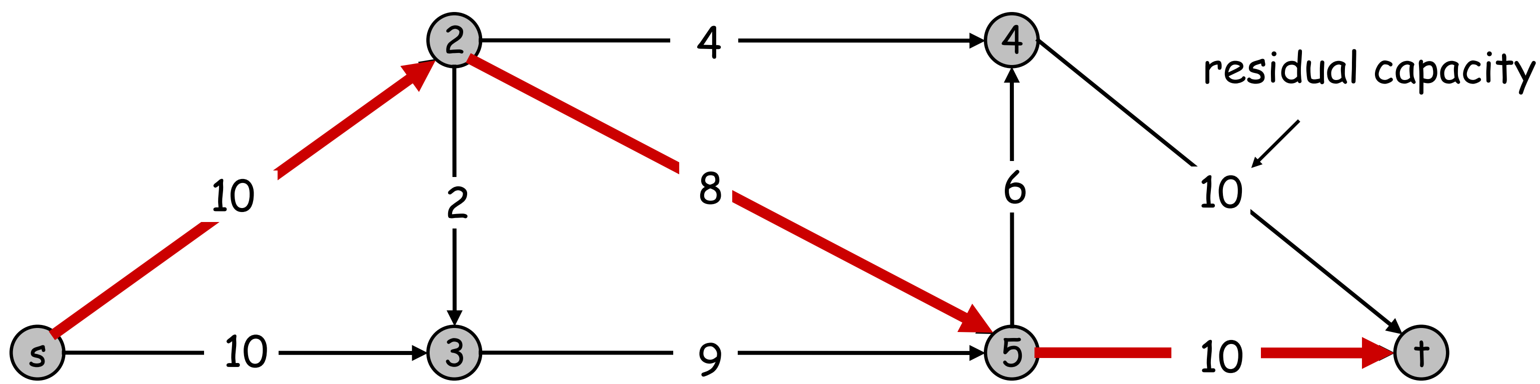
Edmonds-Karp Algorithm

G:

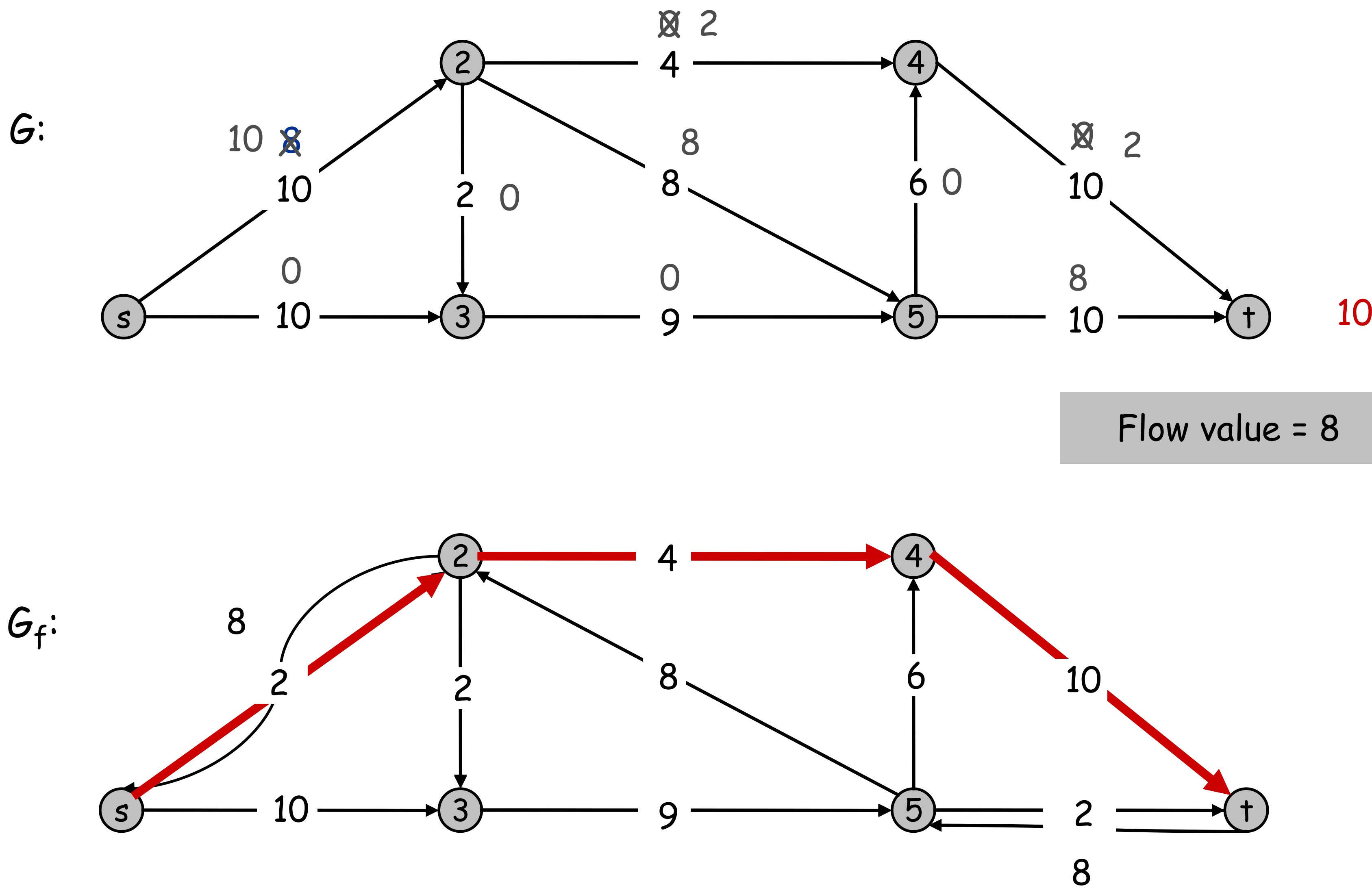


Flow value = 0

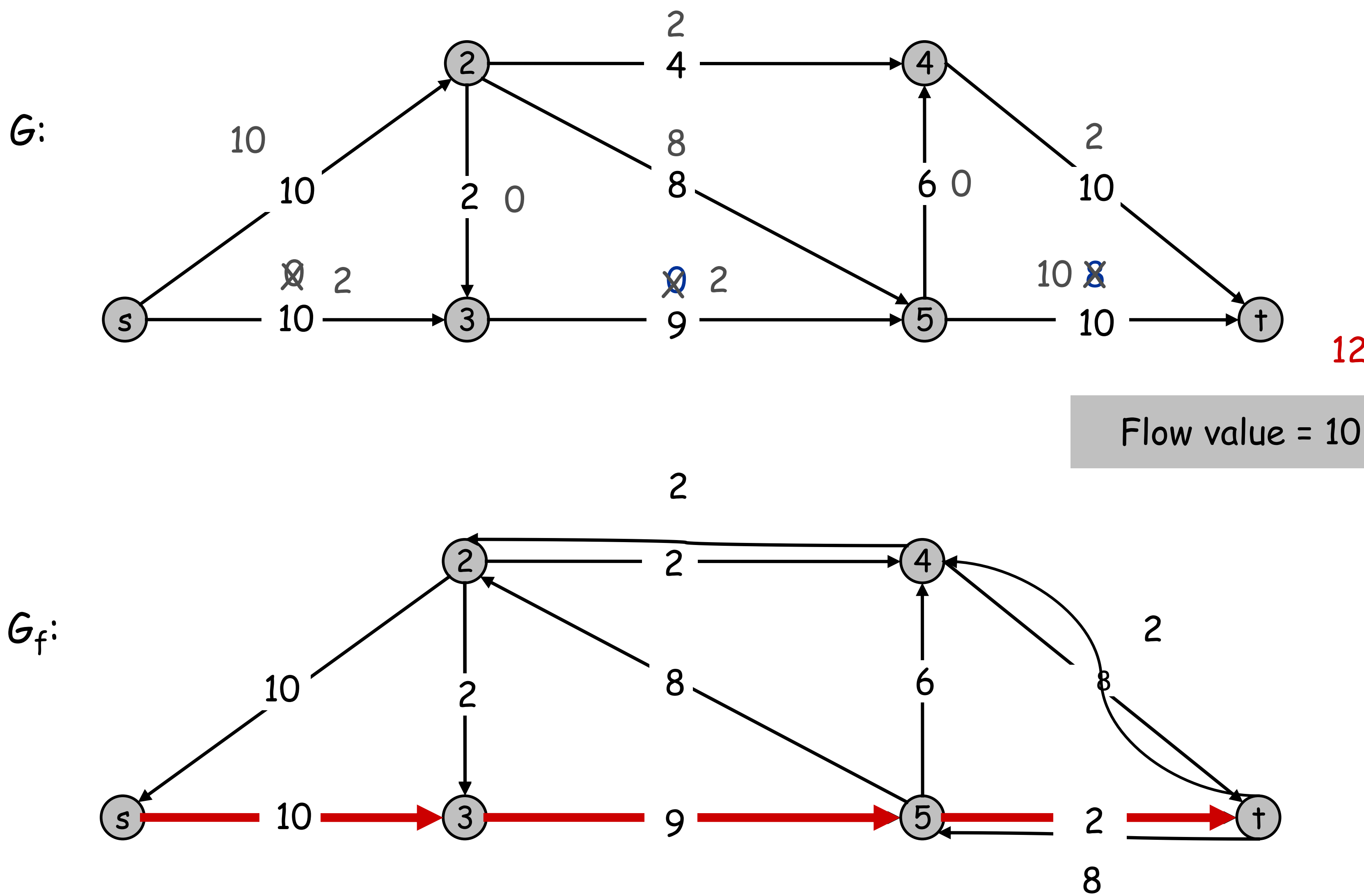
G_f:



Edmonds-Karp Algorithm

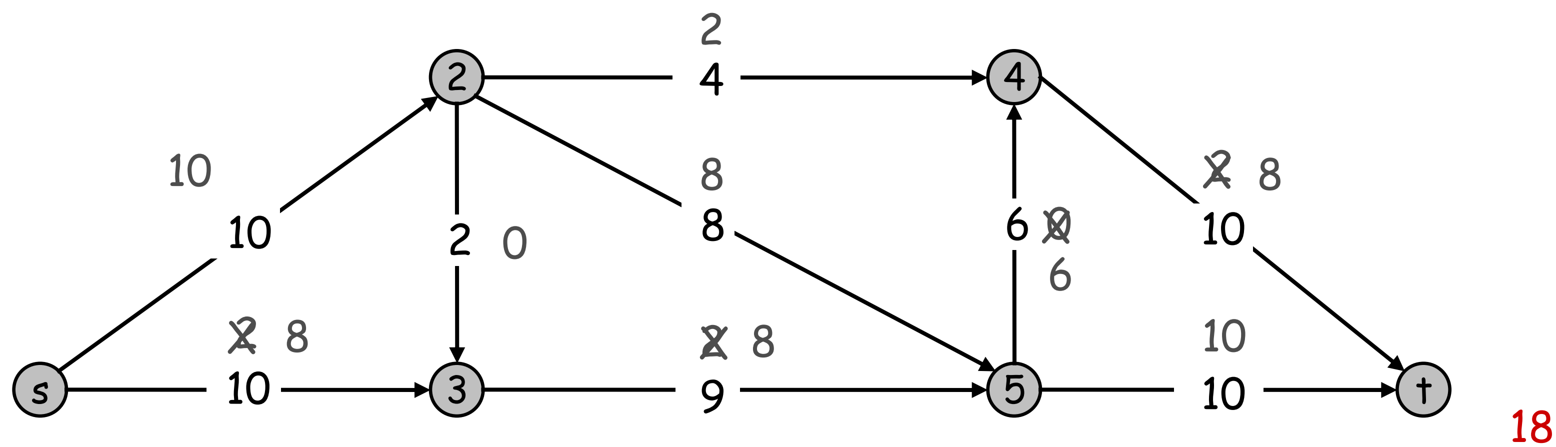


Flow value = 8



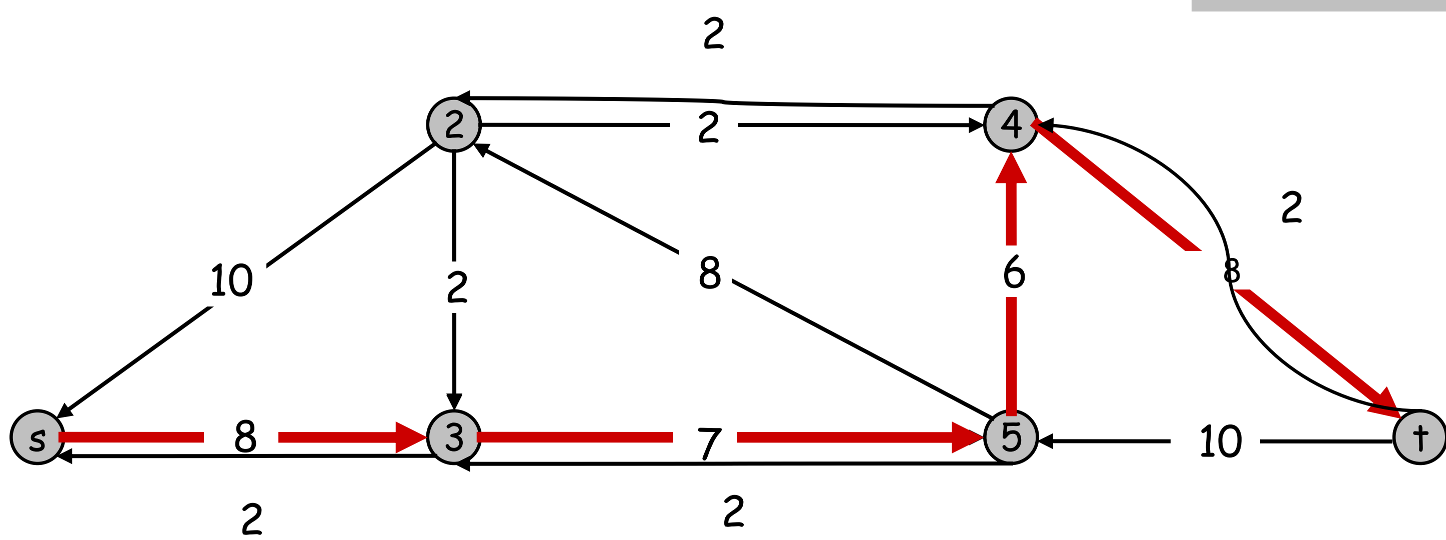
Flow value = 10

G:

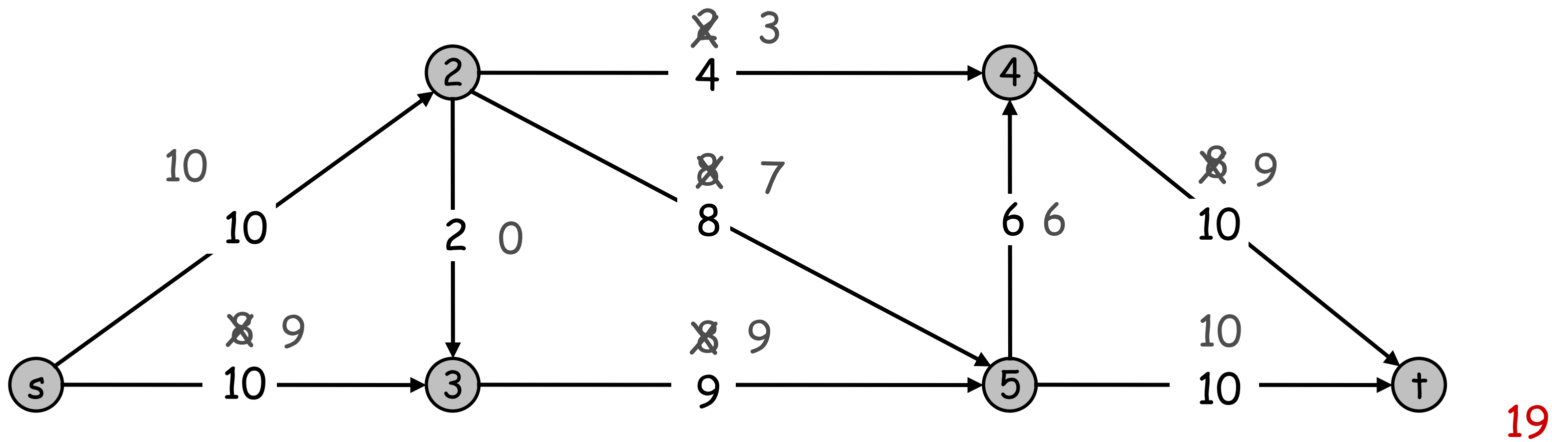


Flow value = 12

G_f:

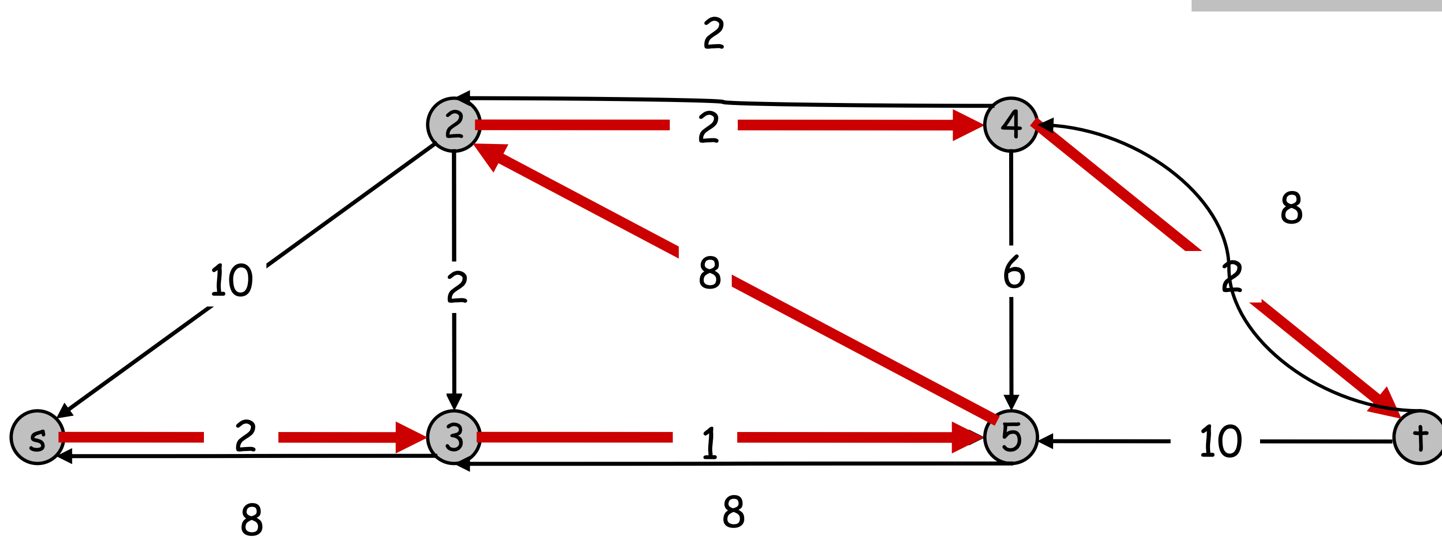


G:

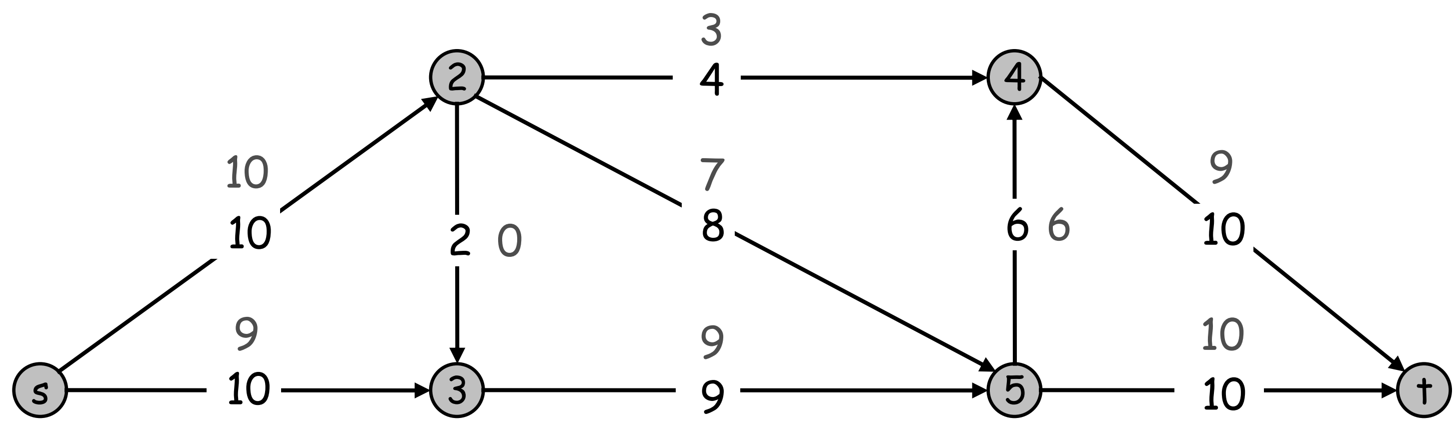


Flow value = 18

G_f:

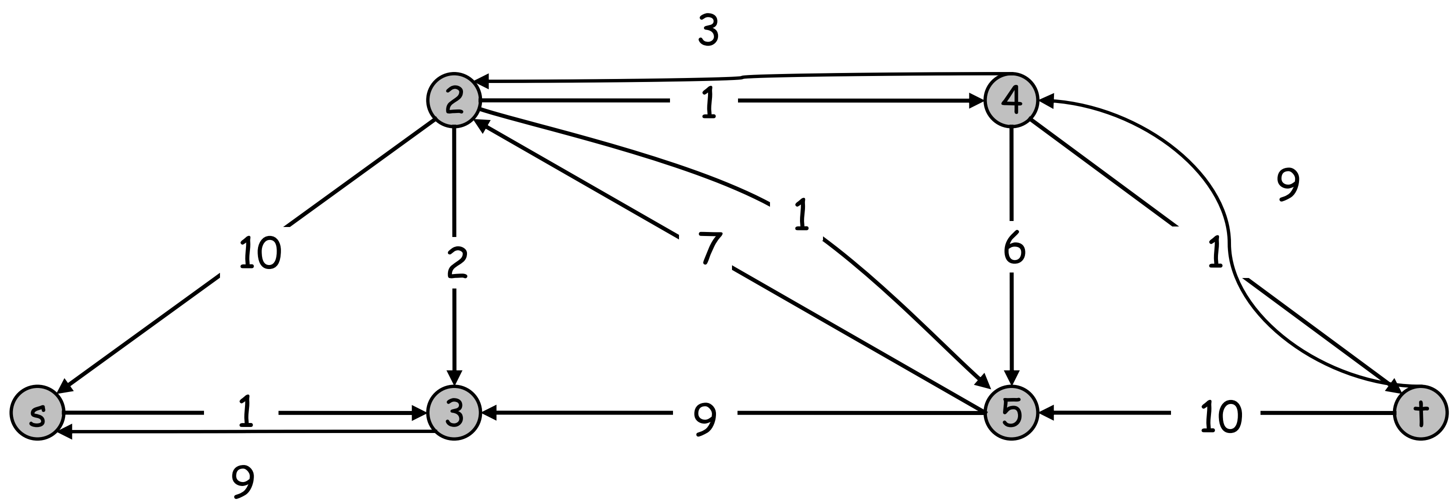


G:



Flow value = 19

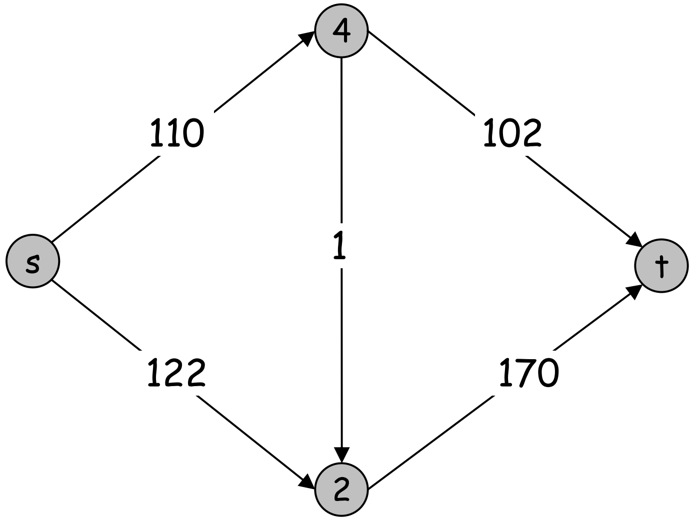
G_f:



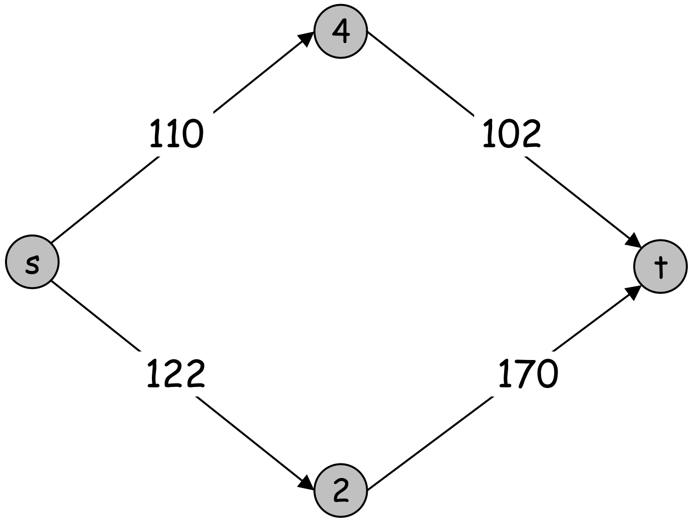
Another way to choose paths - Capacity Scaling

Intuition. Choosing path with highest bottleneck capacity increases flow by max possible amount.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of only arcs with capacity at least Δ.



G_f



$G_f(100)$

Capacity Scaling

```
Scaling-Max-Flow(G, s, t, c) {
   foreach e ∈ E  f(e) ← 0
   Δ ← smallest power of 2 greater than or equal to c(s)
   Gf ← residual graph

   while (Δ ≥ 1) {
      Gf(Δ) ← Δ-residual graph
      while (there exists augmenting path P in Gf(Δ)) {
         f ← augment(f, c, P)
         update Gf(Δ)
      }
      Δ ← Δ / 2
   }
   return f
}
```

$v(f) + E\,\Delta$

$1 + \lceil \log_2 C \rceil$

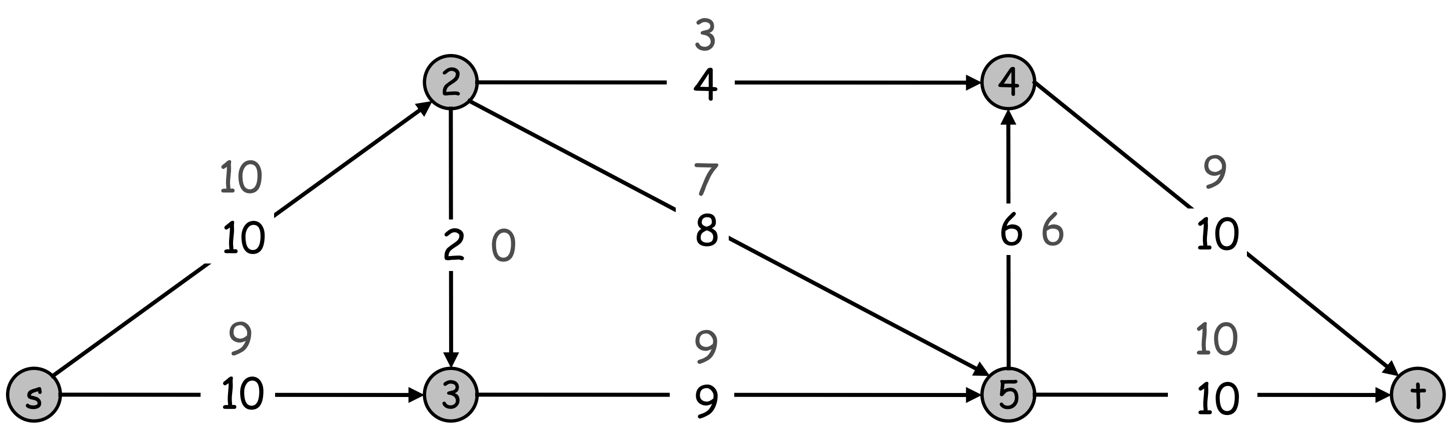
$|E|$

The scaling max-flow algorithm finds a max flow in $O(E \log C)$ augmentations.
It can be implemented to run in $O(E^2 \log C)$ time.

To find the minimum cut

1. Create the maximum flow graph
2. Select all the nodes that can be reached from the source by unsaturated edges
3. Cut all the edges that connect these nodes to the rest of the nodes in the graph
4. This cut will be minimal

G:

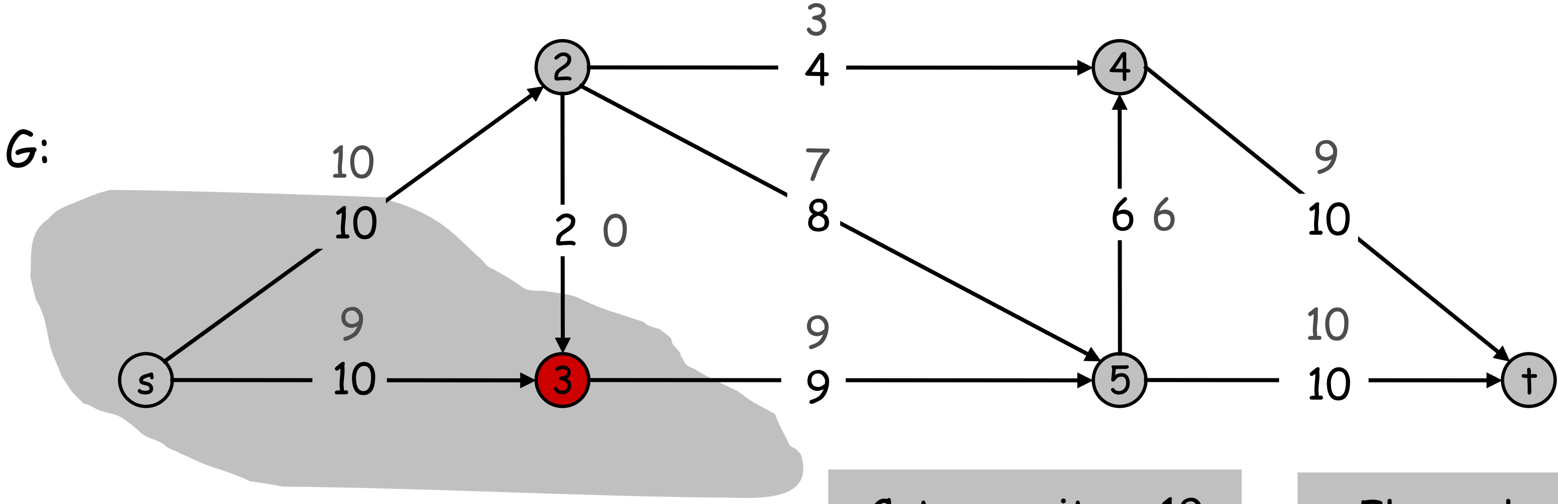


Cut capacity = 19

Flow value = 19

To find the minimum cut

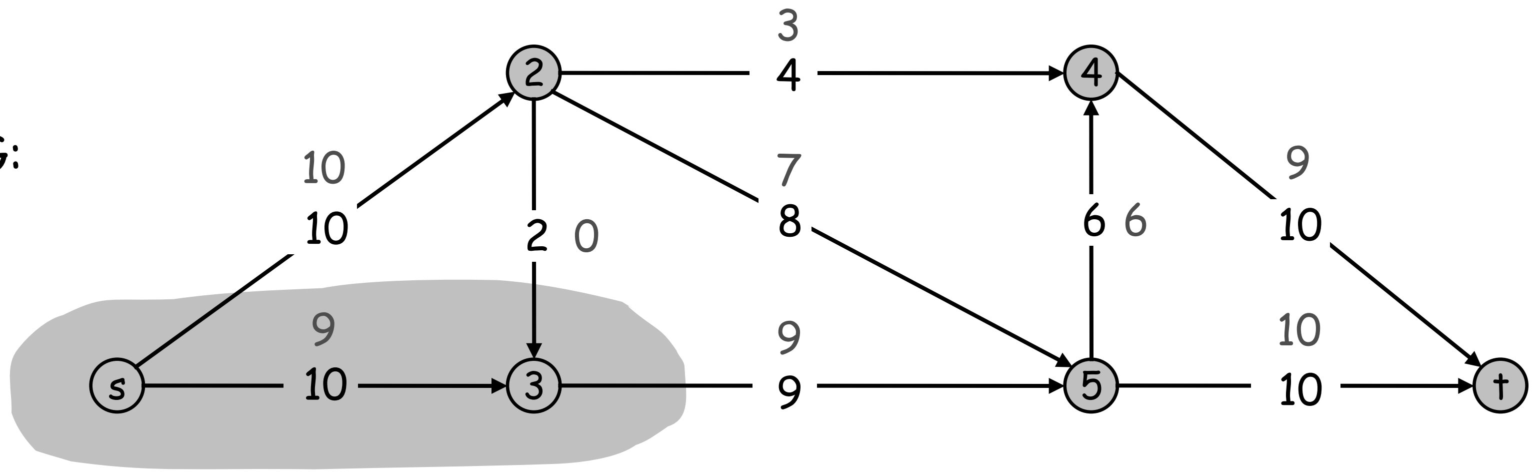
1. Create the maximum flow graph
2. Select all the nodes that can be reached from the source by unsaturated edges
3. Cut all the edges that connect these nodes to the rest of the nodes in the graph
4. This cut will be minimal

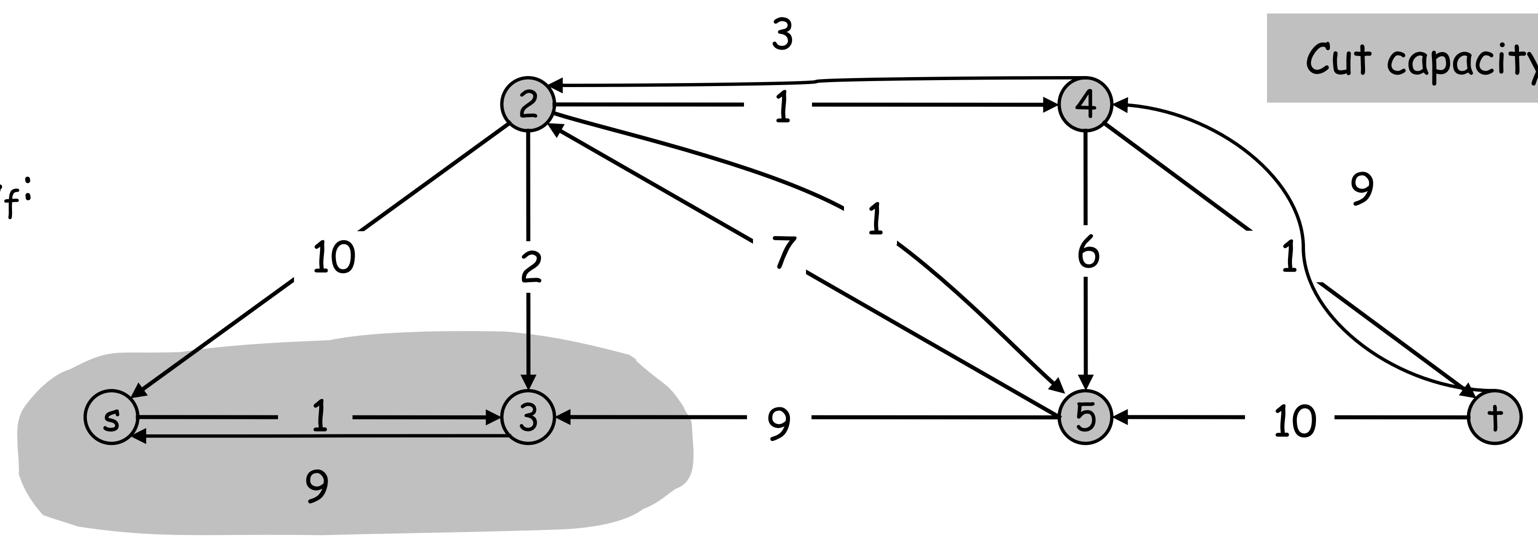


Cut capacity = 19

Flow value = 19

G:





G_f:

Flow value = 19

Cut capacity = 19

History of Algorithms

Augmenting Paths based algorithms

- Ford-Fulkerson (1962) O(E C) -> C is the max capacity
- Edmonds-Karp (1969) $O(VE^2)$

Push-Relabel based algorithms

- Goldberg (1985) $O(V^3)$
- Goldberg and Tarjan (1986) $O(VE\log(V^2/E))$
- Ahuja and Orlin O (1989) $(VE + V^2\log(C))$

Push-relabel algorithms

Flows vs. Preflows

Augmenting Path Algorithm

Flow into i = Flow out of i

Push flow along a path from s to t

$d(j)$ = distance from j to t in the residual network.

Preflow Algorithm

Flow into i ≥ Flow out of i for $i \neq s$.

Push flow in one arc at a time

$d(j) \leq$ distance from j to t in the residual network

- $d(t) = 0$
- $d(i) \leq d(j) + 1$ for each arc $(i, j) \in G(x)$,

From now on, d = h

Preflows

At each intermediate stages we permit more flow arriving at nodes than leaving (except for s)

A *preflow* is a function $x: A \,\square\, R$ s.t. $0 \;\delta\; x \;\delta\; u$ and such that

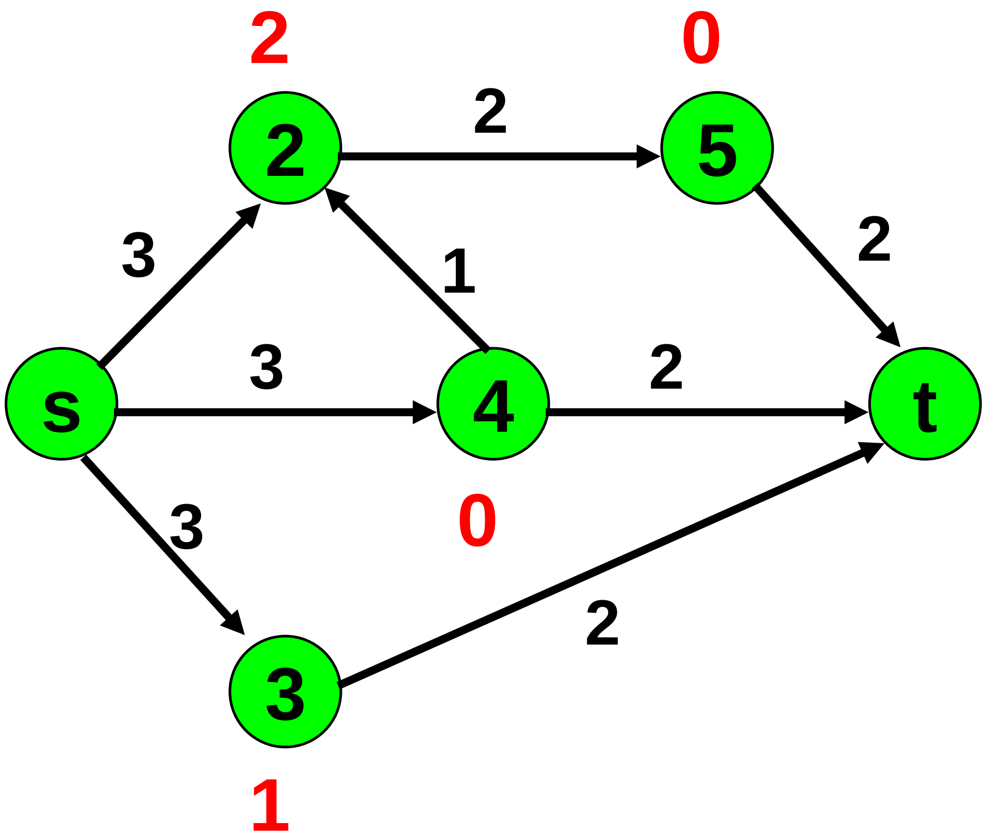
$$e(i) = \sum_{j \in N} x_{ji} - \sum_{j \in N} x_{ij} \geq 0,$$

for all $i \quad N - \{s, t\}$.

i.e., $e(i)$ = *excess* at i = net excess flow into node i.

The excess is required to be nonnegative.

A Feasible Preflow



The *excess* *e(j)* at each node j ≠ s, t is the flow in minus the flow out.

Intuition

- Starting with a preflow, push excess flow closer towards sink
- If excess flow cannot reach sink, push it backwards to source
- Has two main operations:
  - Push
  - Relabel

Residual Graph

Residual capacity $r_f(v, w)$ of a vertex pair is $c(v, w) - f(v, w)$

If:

v has positive excess and

(v,w) has residual capacity,

can push $\delta = \min(e(v), r_f(v, w))$ flow from v to w

Edge (v,w) is *saturated* if $r_f(v, w) = 0$

Residual graph $G_f = (V, E_f)$ where

E_f is the set of *residual edges* (v,w) with $r_f(v, w) > 0$

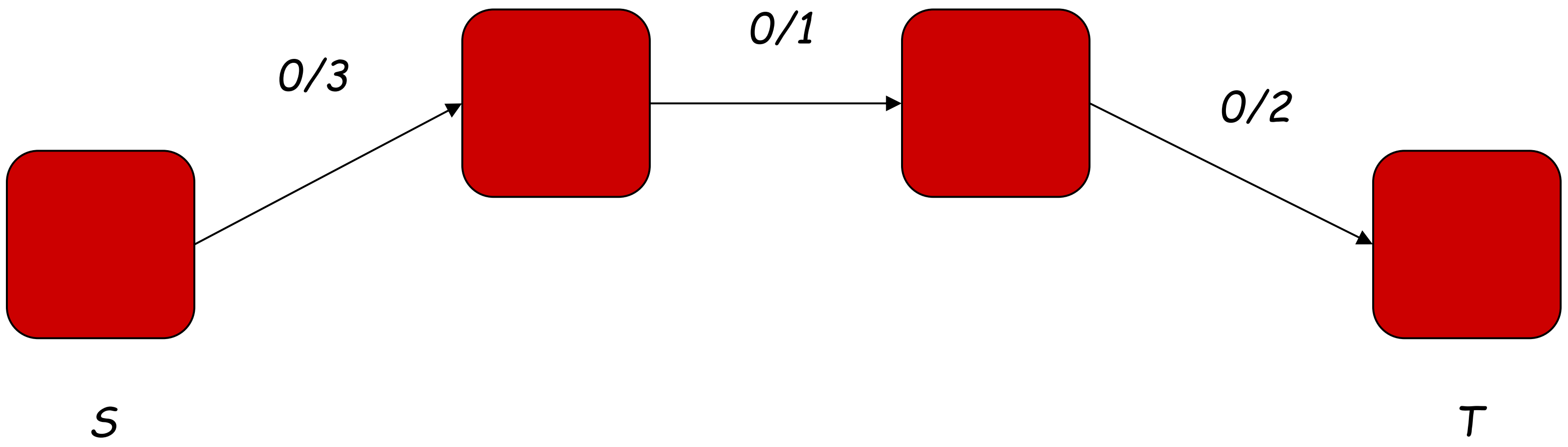
Generic Push-Relabel Algorithm

Starting from an initial preflow

While there is an active vertex

Chose an active vertex v

Apply Push(v,w) for some w or Relabel(v)



Labeling

A valid *labeling* is a function d from vertices to nonnegative integers

- h(s) = |V|
- h(t) = 0
- h(v) <= h(w) + 1 for every residual edge

If h(v) < n, h(v) is a lower bound on distance to sink
If h(v) >= n, h(v) - n is a lower bound on distance to source

Push and Relabel Operations

Push(v,w)

Precondition: v is active ($e(v) > 0$)

$r_f(v,w) > 0$

$v.h = w.h + 1$

Action: Push $\delta = \min(e(v), r_f(w, v))$ from u to v

$f(v,w) = f(v,w) + \delta;$

$f(w,v) = f(w,v) – \delta;$

$e(v) = e(v) - \delta;$

$e(w) = e(w) + \delta;$

Relabel(v)

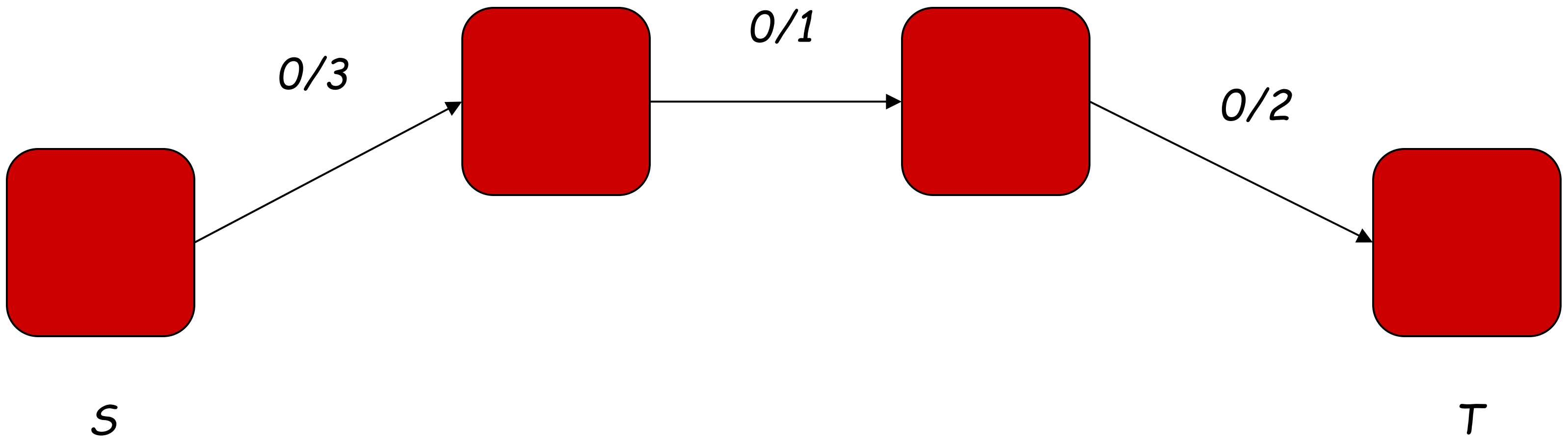
Precondition: v is active ($e(v) > 0$)

$r_f(v, w) > 0$ implies $v.h <= w.h$

Action: $v.h = 1+ \min\{w.h \mid (v,w) \in E_f\}$

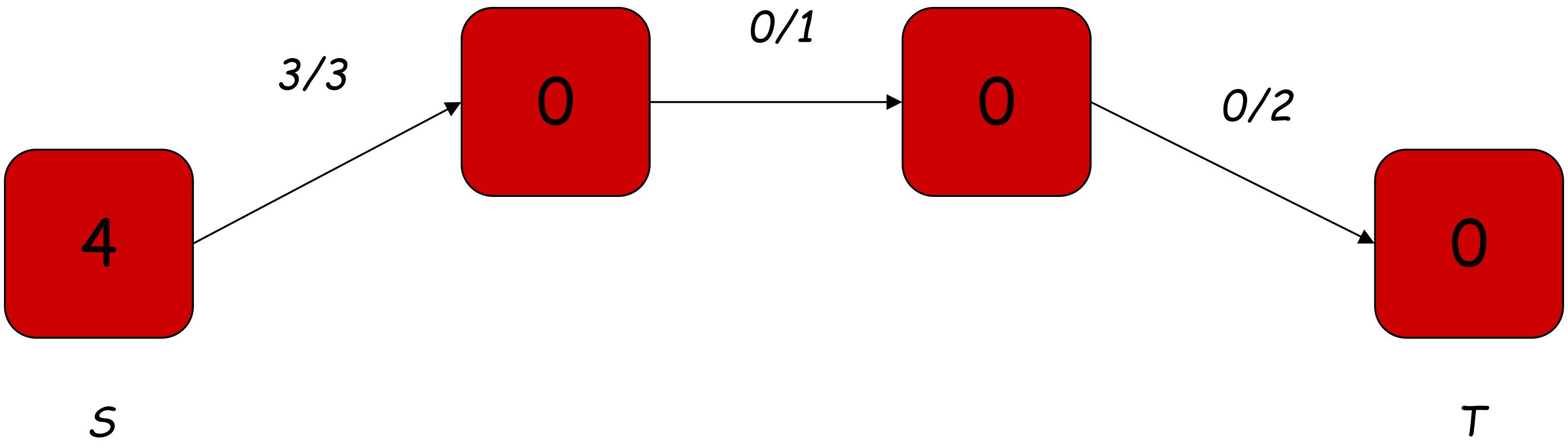
Initialize_Preflow

```
for each vertex v in G
     v.h = 0      Distance
     v.e = 0      Excess
for each edge (u,v) in E
     (u.v).f = 0
s.h = |G.V|
for each vertex ∊ s.adj
      (s,v).f = c(s,v)
      v.e = c(s,v)
      s.e = s.e - c(s,v)
```



Initialize_Preflow

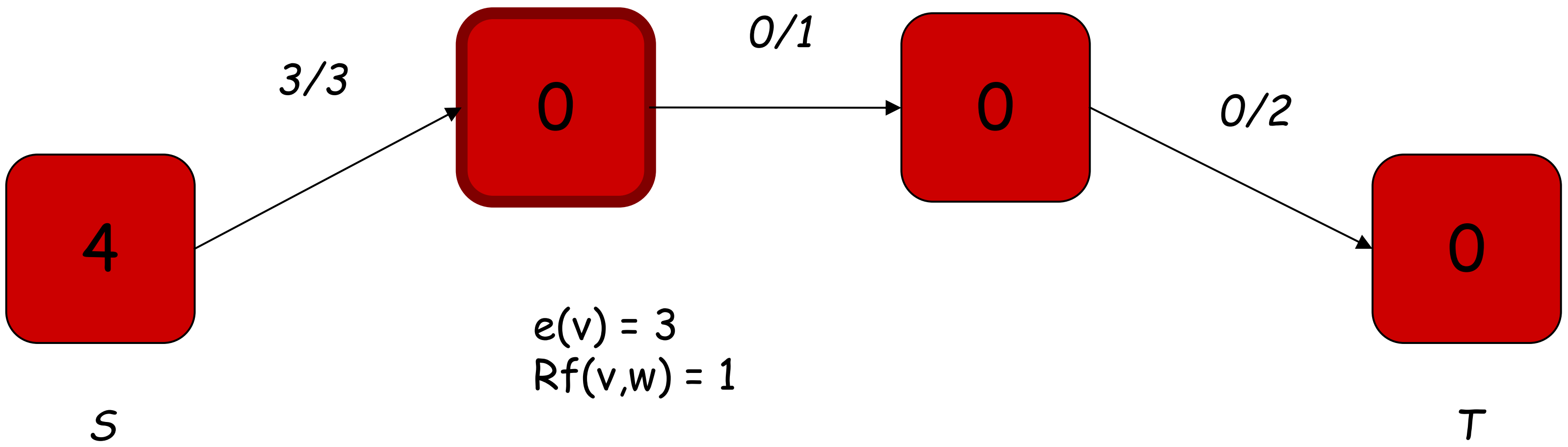
```
for each vertex v in G
     v.d = 0       Distance
      v.e = 0      Excess
for each edge (u,v) in E
     (u.v).f = 0
s.h = |G.V|
for each vertex ∈ s.adj
     (s.v).f = c(s,v)
      v.e = c(s,v)
      s.e = s.e-c(s,v)
```



Example

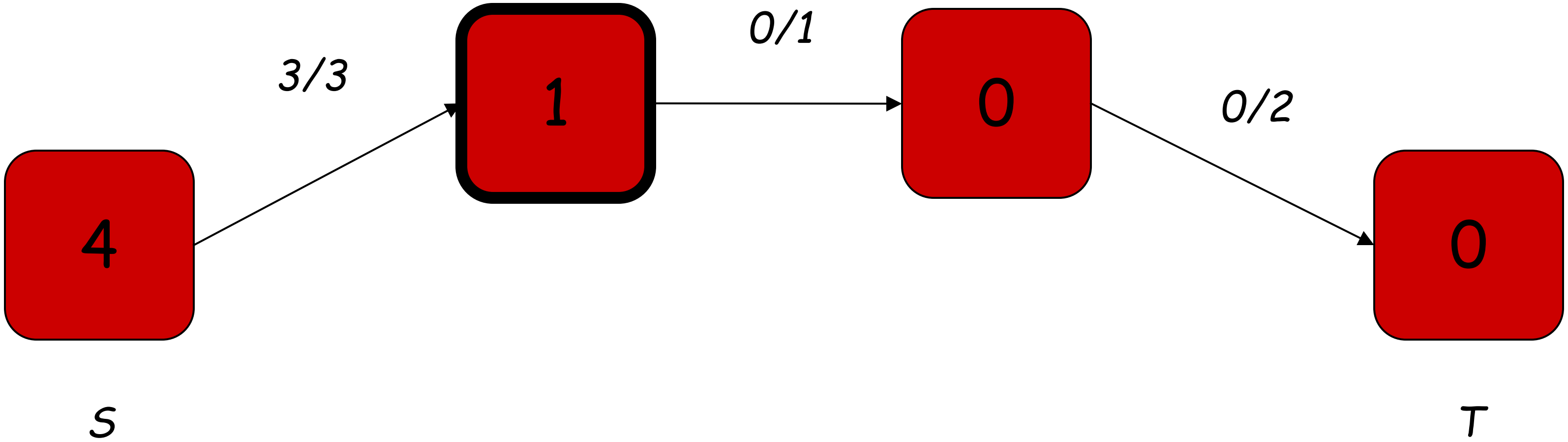
Push(v,w)
v is active if ($e(v) > 0$)
$r_f(v,w) > 0$
$v.h = w.h + 1$

Relabel(v)
v is active ($e(v) > 0$)
$r_f(v,w) > 0$ implies $v.h <= w.h$



Select an active vertex

Example

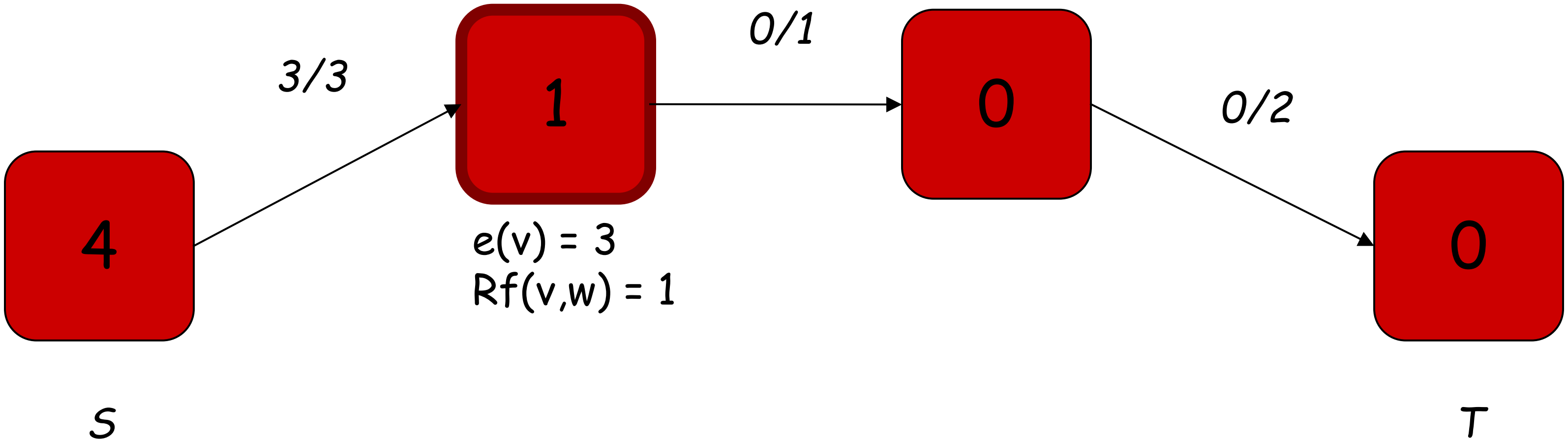


Relabel active vertex

Example

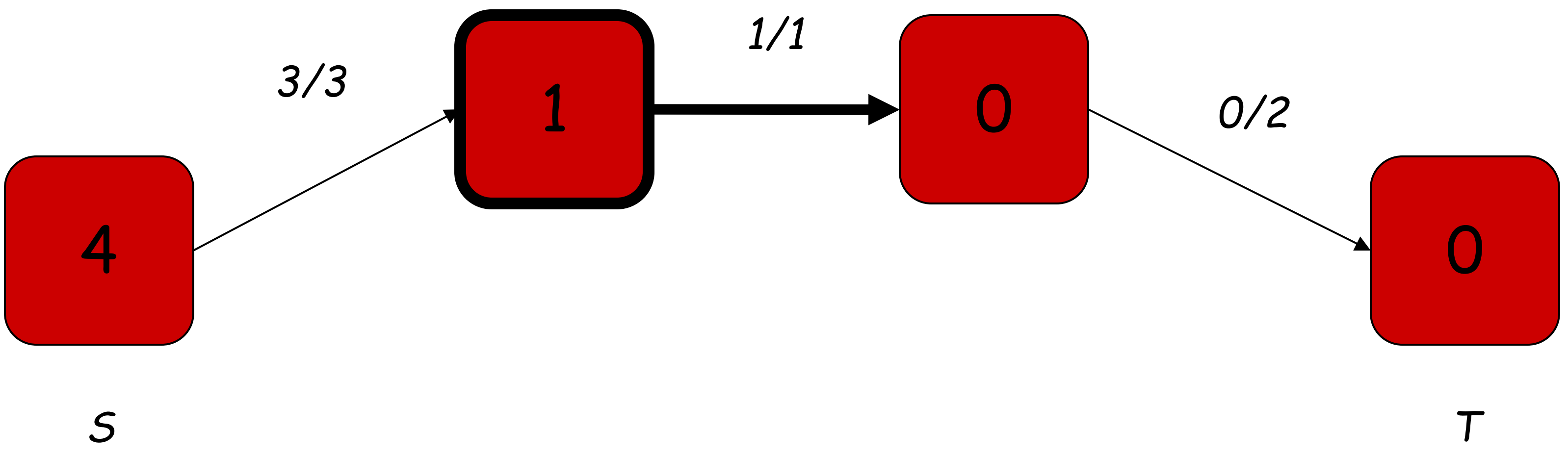
Push(v,w)
v is active if (e(v) > 0)
$r_f(v,w) > 0$
$v.d = w.d + 1$

Relabel(v)
v is active (e(v) > 0)
$r_f(v,w) > 0$ implies $v.h <= w.h$



Select an active vertex

Example

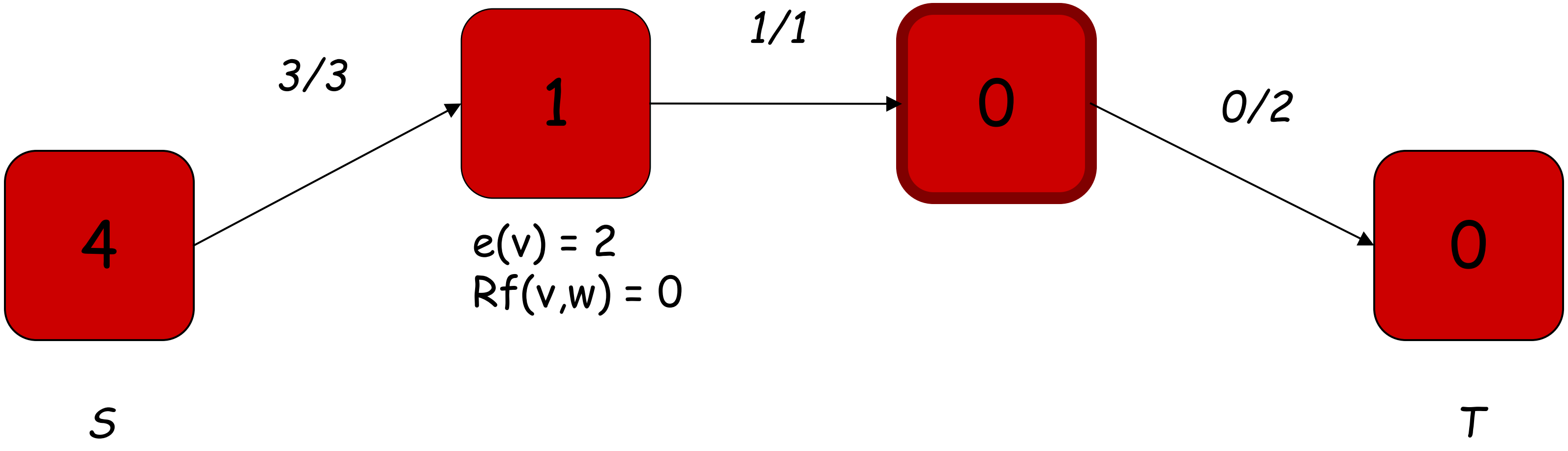


Push excess from active vertex

Example

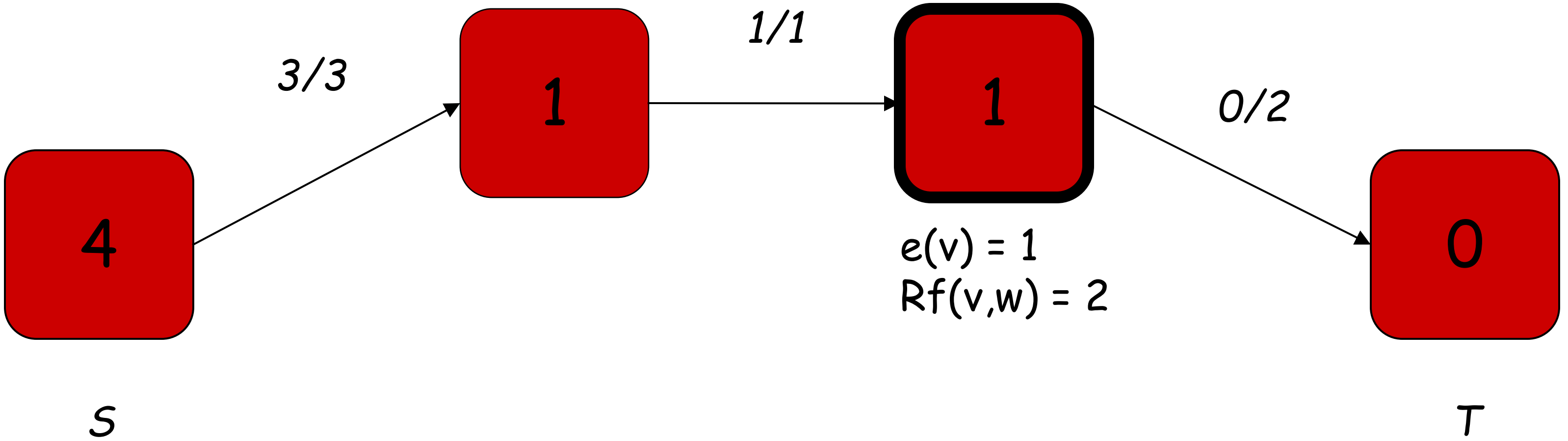
Push(v,w)
v is active if $(e(v) > 0)$
$r_f(v,w) > 0$
$v.d = w.d + 1$

Relabel(v)
v is active $(e(v) > 0)$
$r_f(v,w) > 0$ implies $v.h <= w.h$



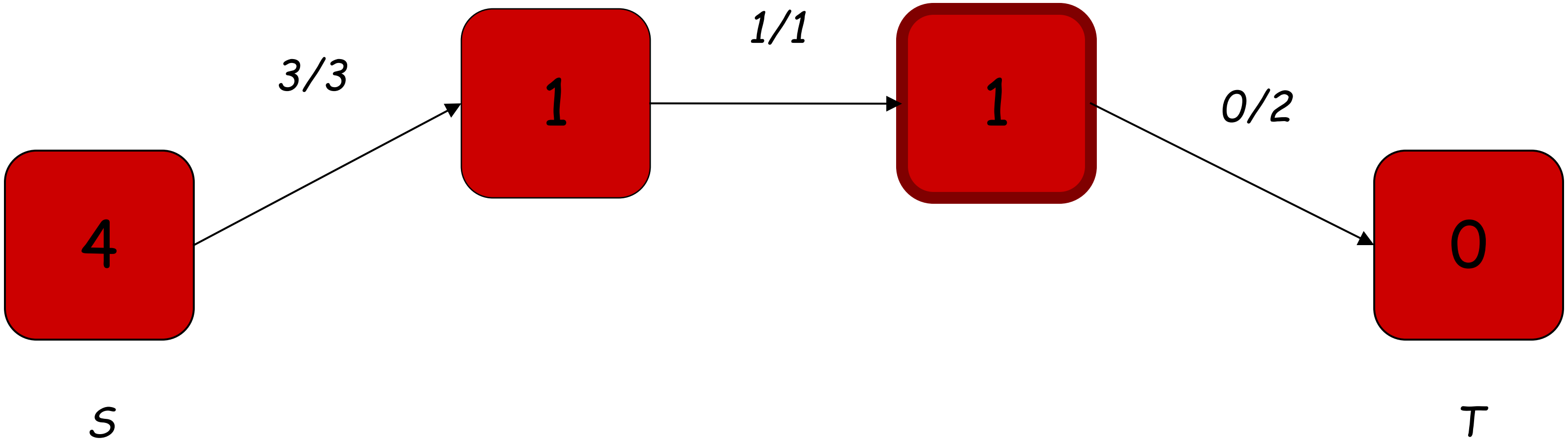
Select an active vertex

Example



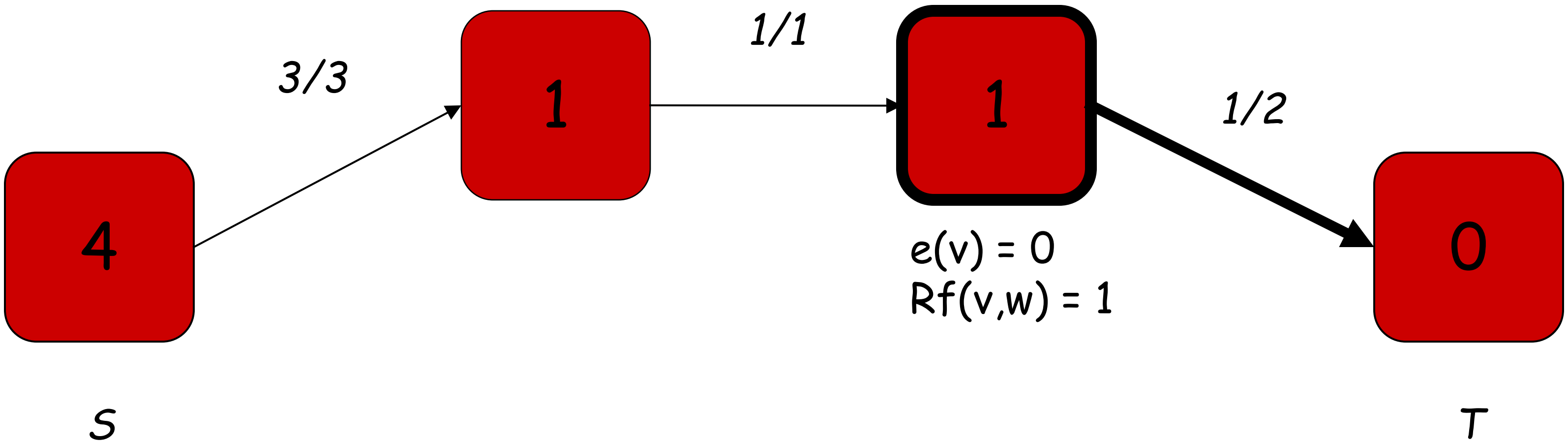
Relabel active vertex

Example



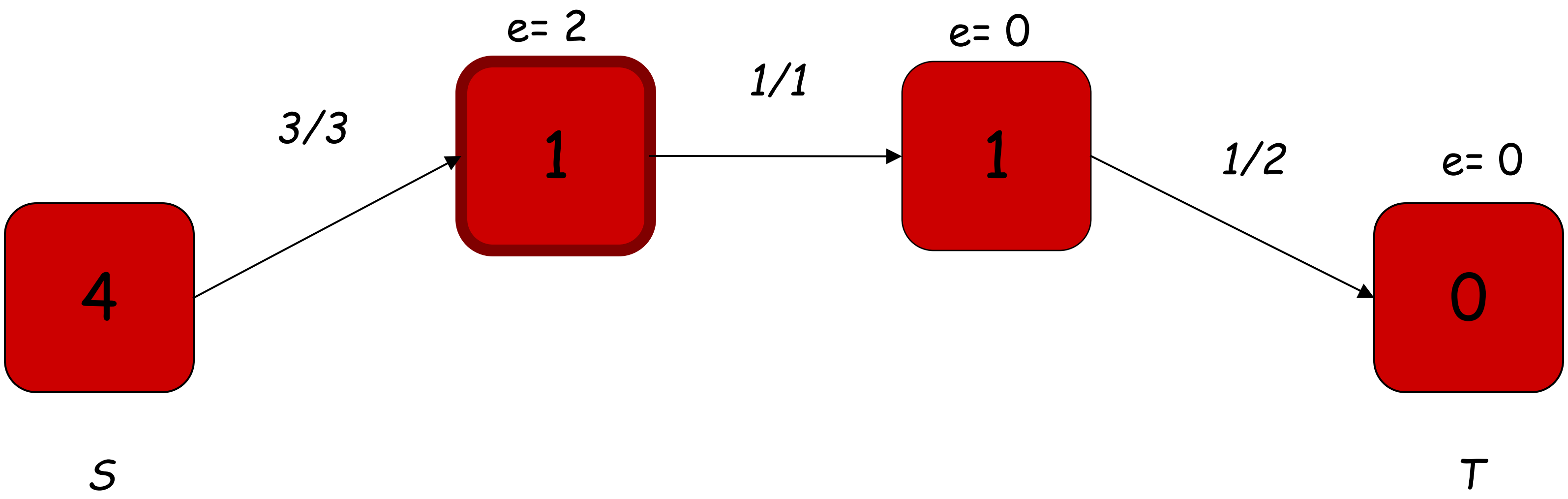
Select an active vertex

Example



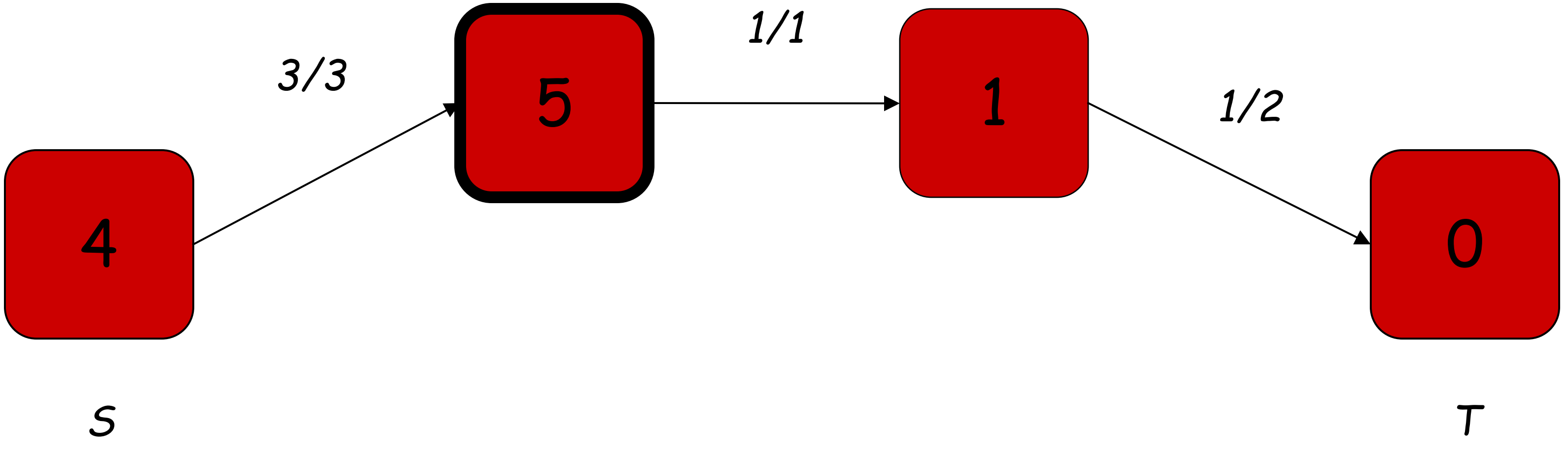
Push excess from active vertex

Example



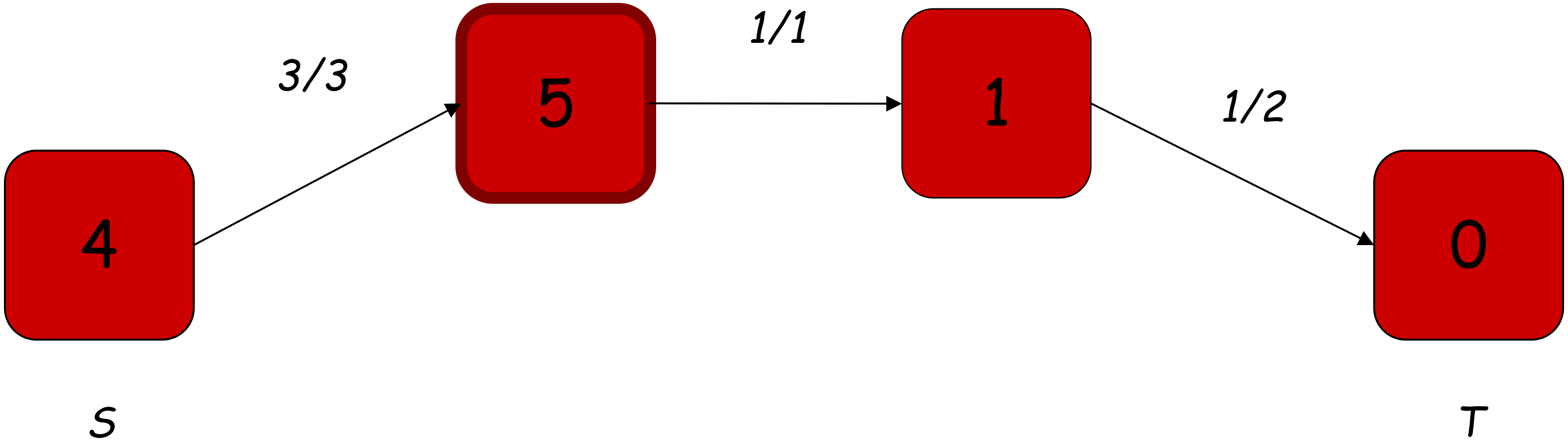
Select an active vertex

Example



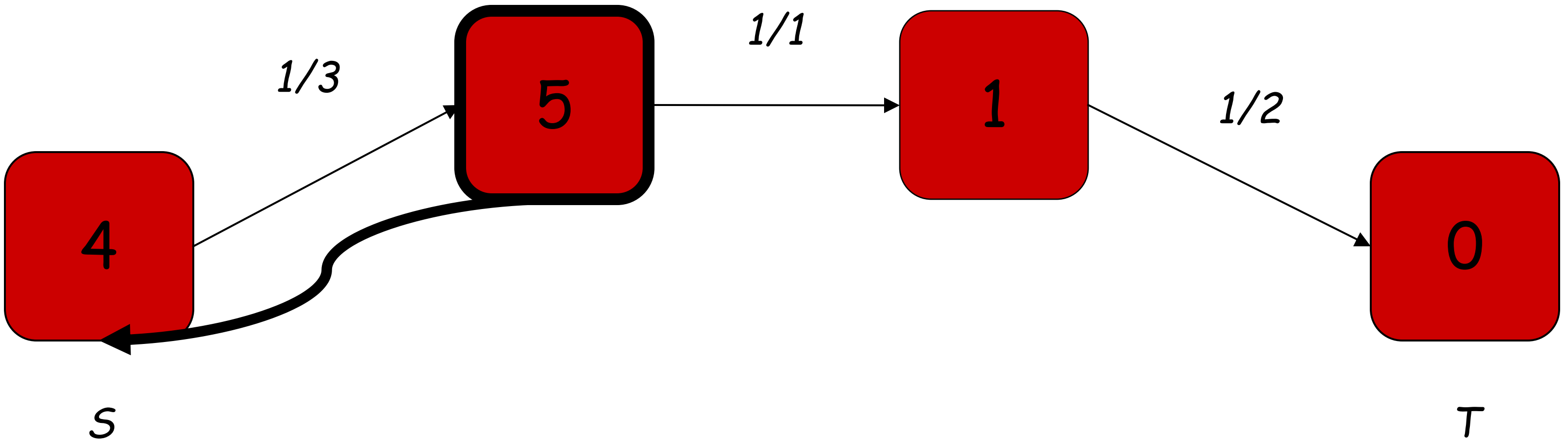
Relabel active vertex

Example



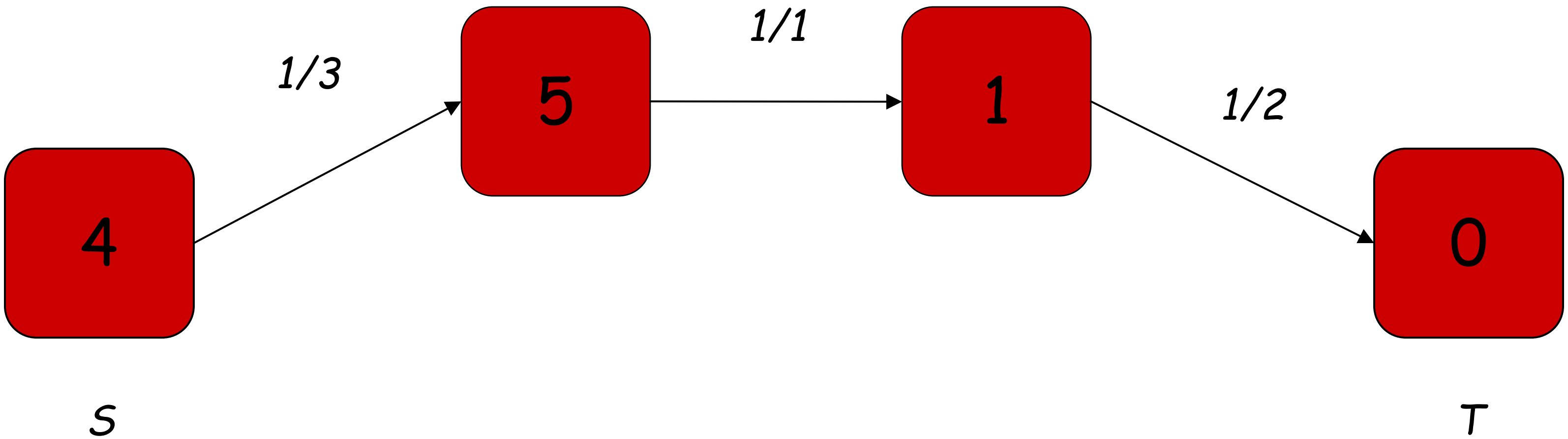
Select an active vertex

Example



Push excess from vertex

Example



Maximum flow

Research notes on preflow push

- Pushing from the active node with the largest distance label leads to $O(n^2 m^{.5})$ nonsaturation pushes.
- A very efficient data structure called dynamic trees reduces the running time to $O(nm \log n^2/m)$. Goldberg-Tarjan (1986)
- The “excess scaling technique” of Ahuja and Orlin (1989) reduced the running time to $O(nm + n^2 \log U)$.
- Ahuja, Orlin, and Tarjan (1989): further very small improvements.
- Goldberg and Rao (1998). An even more efficient algorithm for max flows.

Summary

Augmenting Paths based algorithms

- Ford-Fulkerson (1962) $O(E\ C)$ -> C is the max capacity
- Edmonds-Karp (1969) $O(VE^2)$

Push-Relabel based algorithms

- Goldberg (1985) $O(V^3)$
- Goldberg and Tarjan (1986) $O(VE\log(V^2/E))$
- Ahuja and Orlin $O(VE + V^2\log(C))$