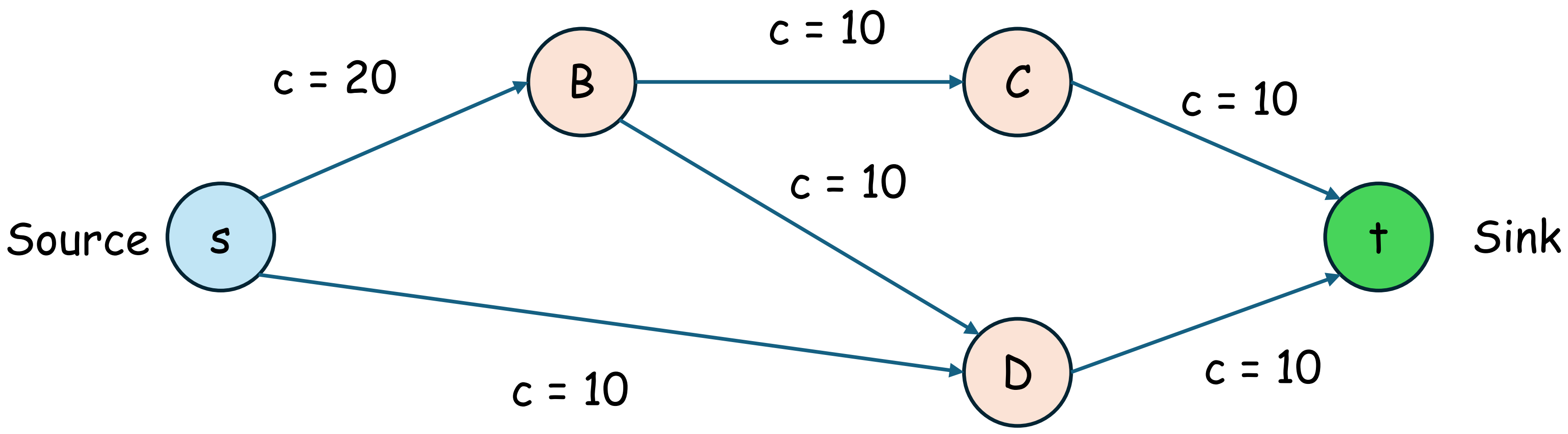
Max-Flow Problem

Fakultas Ilmu Komputer
Universitas Indonesia

Compiled by **Alfan F. Wicaksono** from multiple sources

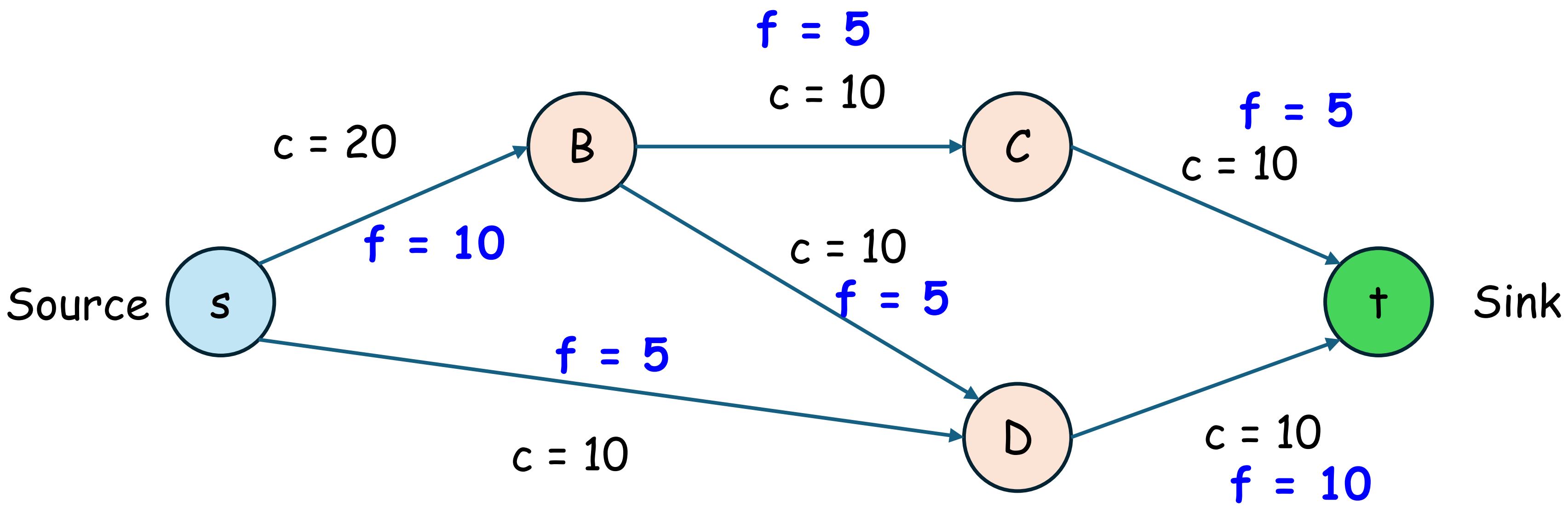
Suppose we consider a **directed** graph **G = (V,E)**, where each edge has a **flow capacity** (how much this edge can carry?); and there is a single **source** node and a **sink** node.



Suppose we consider a **directed** graph **G = (V,E)**, where each edge has a **flow capacity** (how much this edge can carry?);

and there is a single **source** node and a **sink** node.

A **flow** is an "abstract entity" that is generated at a source node, transmitted across edges, and absorbed at a sink node.



Flow Networks

- Assumptions:
  - no edge enters the source s and no edge leaves the sink t;
  - there is at least one edge incident to each node;
  - all capacities are integers.
- These assumptions make things cleaner to think about, and while they eliminate a few pathologies, they preserve all the important issues.

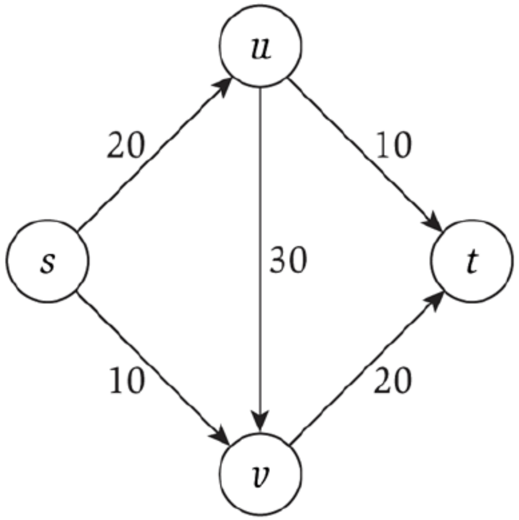


Figure 7.2 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

Flow

- What does it mean for a network to carry traffic, or flow?
- An s-t flow is a function f that maps each edge e to a nonnegative real number, $f : E \rightarrow \mathbf{R}^+$; the value f(e) intuitively represents the amount of flow carried by edge e.
- A flow f must satisfy the following two properties:

  (i) (Capacity conditions) For each $e \in E$, we have $0 \leq f(e) \leq c(e)$.

  (ii) (Conservation conditions) For each node v other than s and t, we have

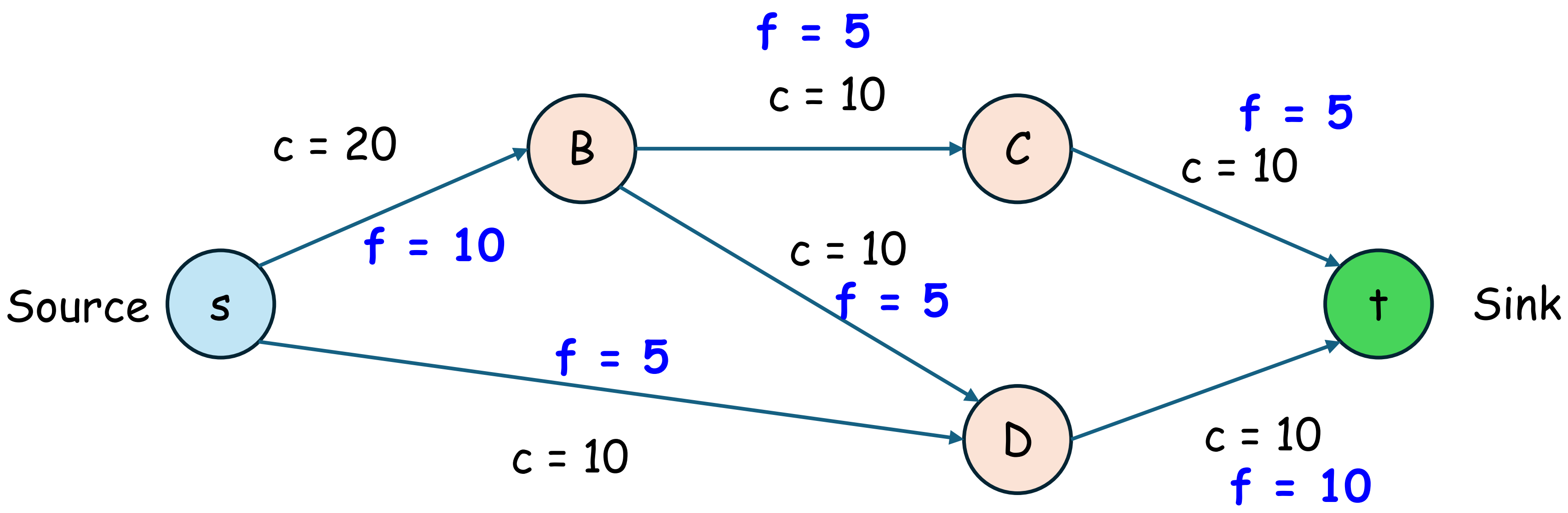
$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

Flow

- The value of a flow f, denoted v(f), is defined to be the amount of flow generated at the source:

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

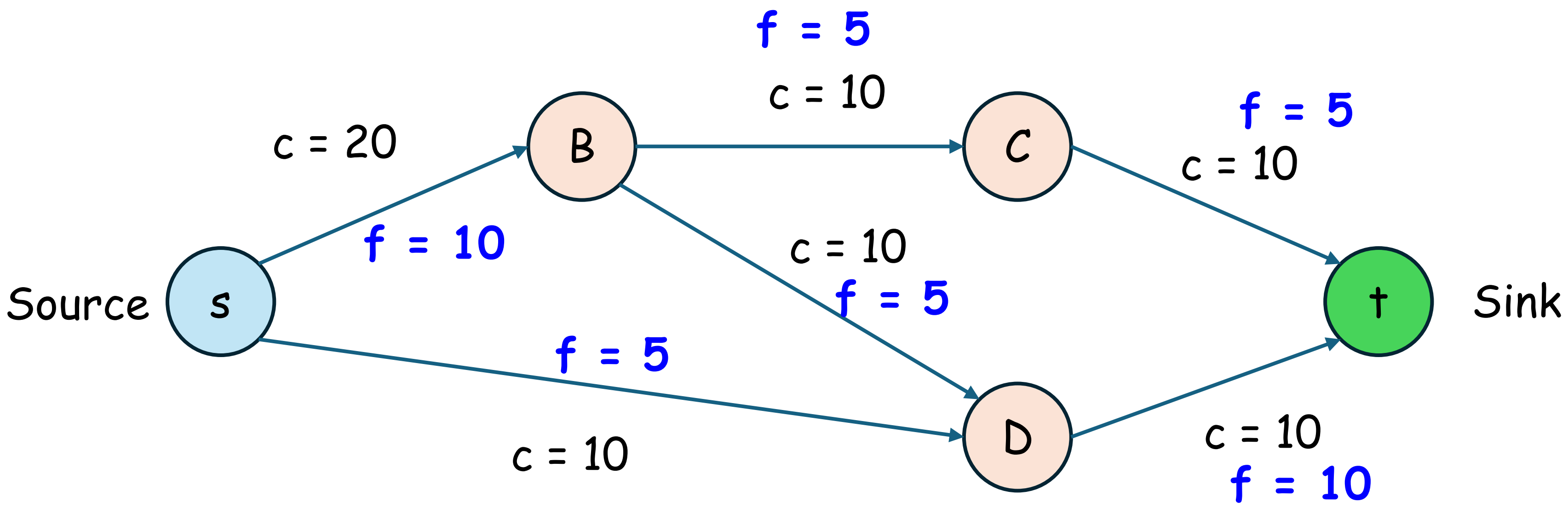
In the following example, v(f) = 10 + 5 = 15



Maximum-Flow Problem

Given a flow network, a goal is to arrange the flow so as to make as efficient use as possible of the available capacity.

--> Goal: **Find a flow of maximum possible value!**

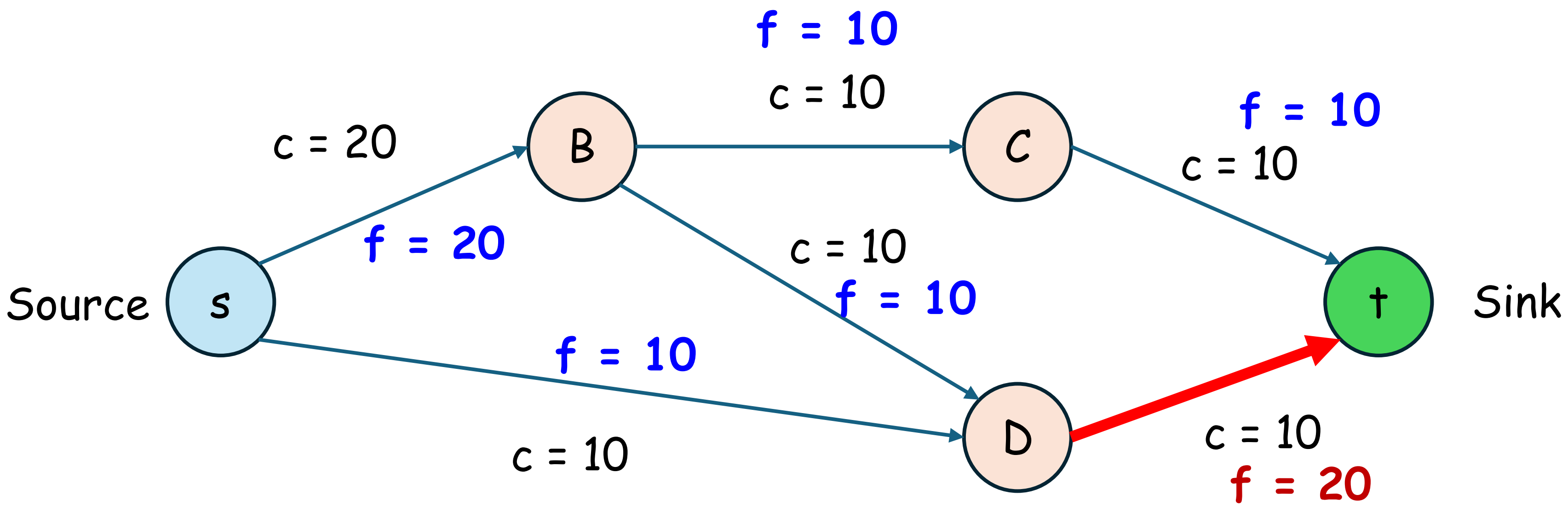


v(f) = 15, is this an optimal flow? **NO**

Maximum-Flow Problem

Given a flow network, a goal is to arrange the flow so as to make as efficient use as possible of the available capacity.

--> Goal: **Find a flow of maximum possible value!**

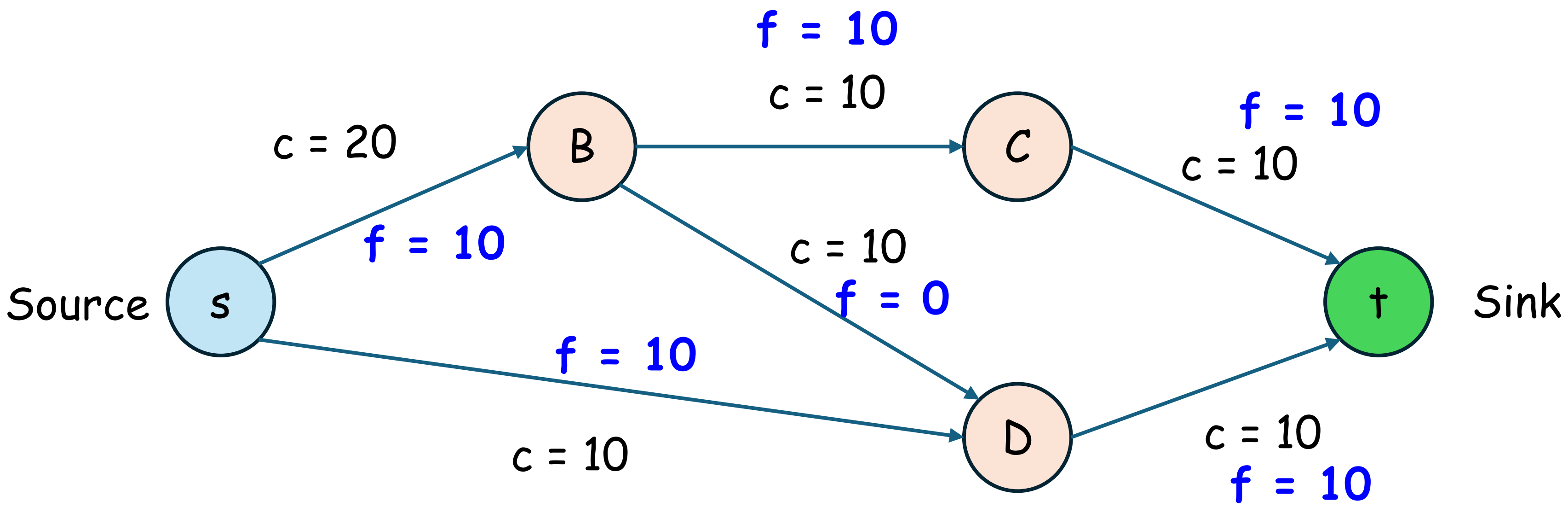


v(f) = 30, is this an optimal flow? **NO … A flow @ edge (D, t) is over capacity!**

Maximum-Flow Problem

Given a flow network, a goal is to arrange the flow so as to make as efficient use as possible of the available capacity.

--> Goal: **Find a flow of maximum possible value!**

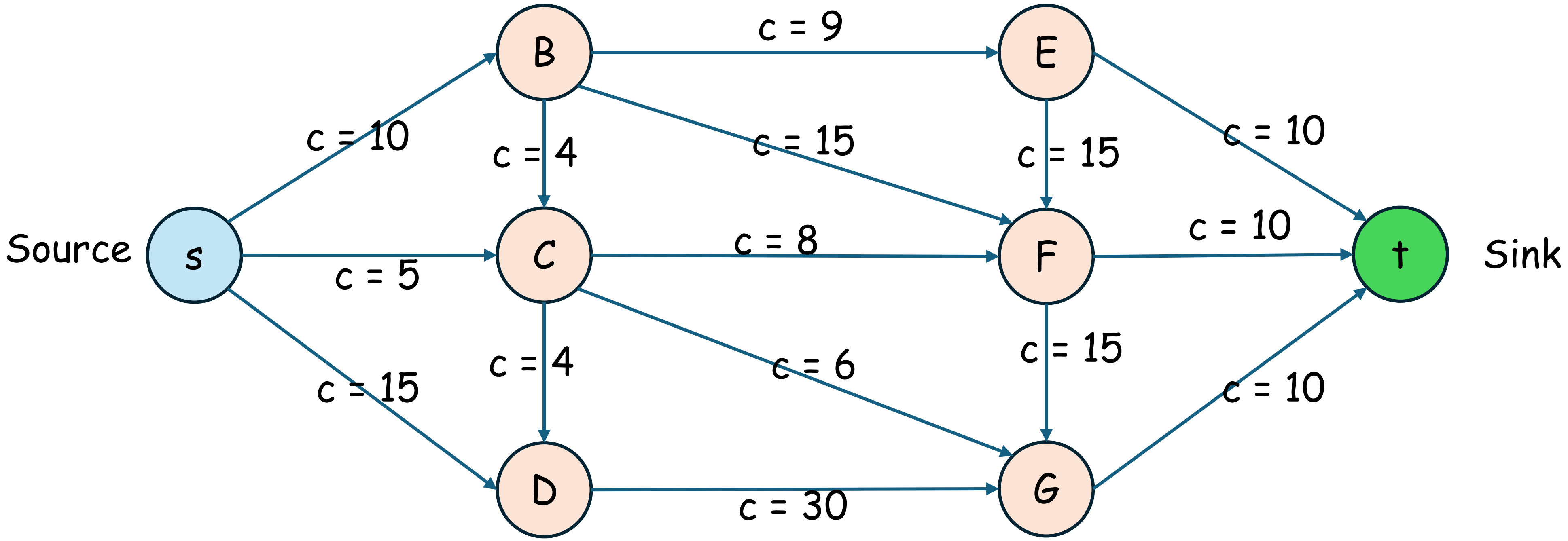


v(f) = 20, is this an optimal flow? **YES**

Maximum-Flow Problem

Given a flow network, a goal is to arrange the flow so as to make as efficient use as possible of the available capacity.

--> Goal: **Find a flow of maximum possible value!**

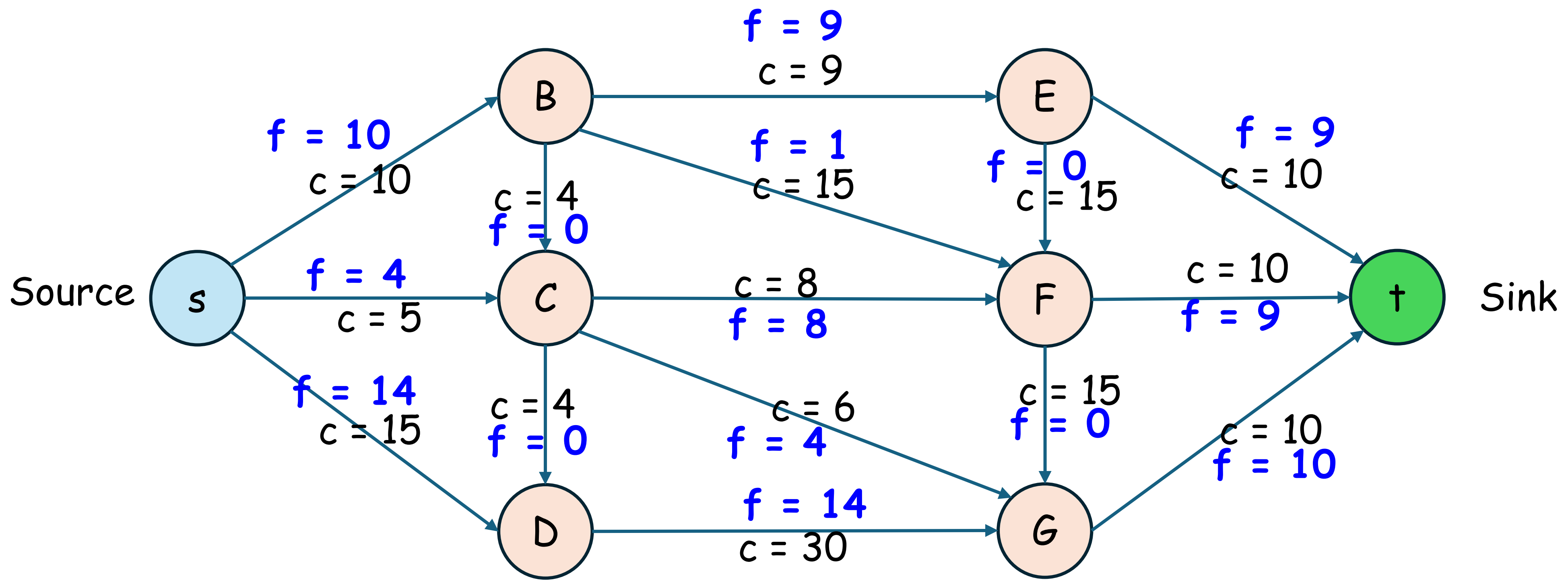


What if the flow graph is a bit more complex ☺, Can you guess?

Maximum-Flow Problem

Given a flow network, a goal is to arrange the flow so as to make as efficient use as possible of the available capacity.

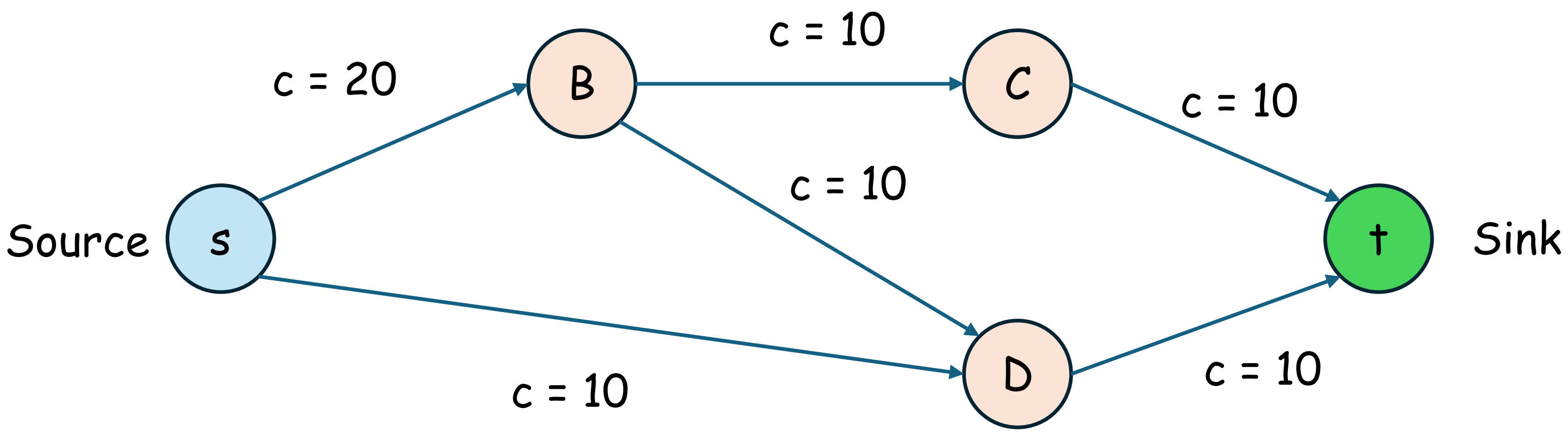
--> Goal: **Find a flow of maximum possible value!**



The maximum flow --> v(f) = 28

upper bounds on an s-t flow

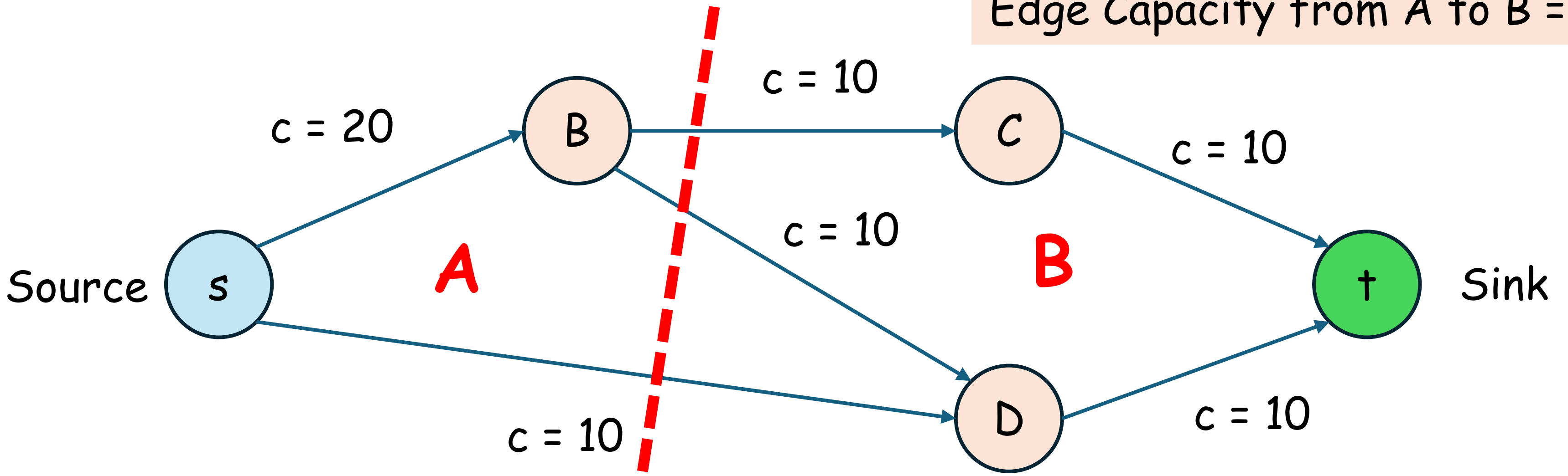
- A basic obstacle to the existence of large flows is as follows: Suppose we divide the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. Then, intuitively, any flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such division (cut) of the graph puts a bound on the maximum possible flow value.
- We will learn later that the maximum-flow value equals the minimum capacity of any such division, called the minimum cut.



upper bounds on an s-t flow

- A basic obstacle to the existence of large flows is as follows: Suppose we divide the nodes of the graph into two sets, A and B, so that s ∈ A and t ∈ B. Then, intuitively, any flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such division (cut) of the graph puts a bound on the maximum possible flow value.
- We will learn later that the maximum-flow value equals the minimum capacity of any such division, called the minimum cut.

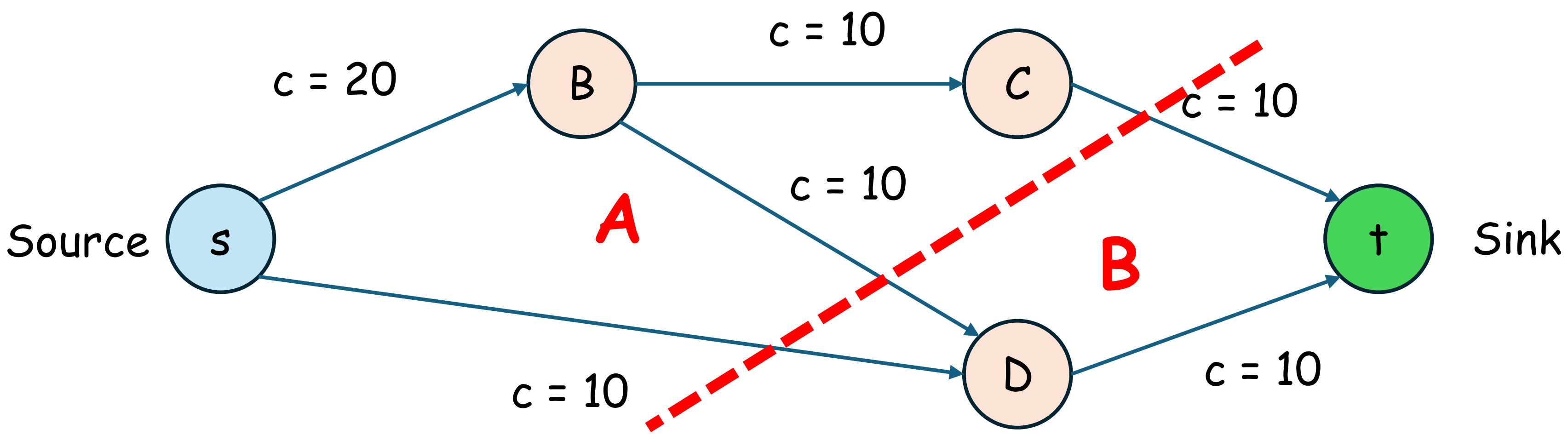
Edge Capacity from A to B = 10 + 10 + 10 = 30



upper bounds on an s-t flow

- A basic obstacle to the existence of large flows is as follows: Suppose we divide the nodes of the graph into two sets, A and B, so that $s \in A$ and $t \in B$. Then, intuitively, any flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such division (cut) of the graph puts a bound on the maximum possible flow value.
- We will learn later that the maximum-flow value equals the minimum capacity of any such division, called the minimum cut.

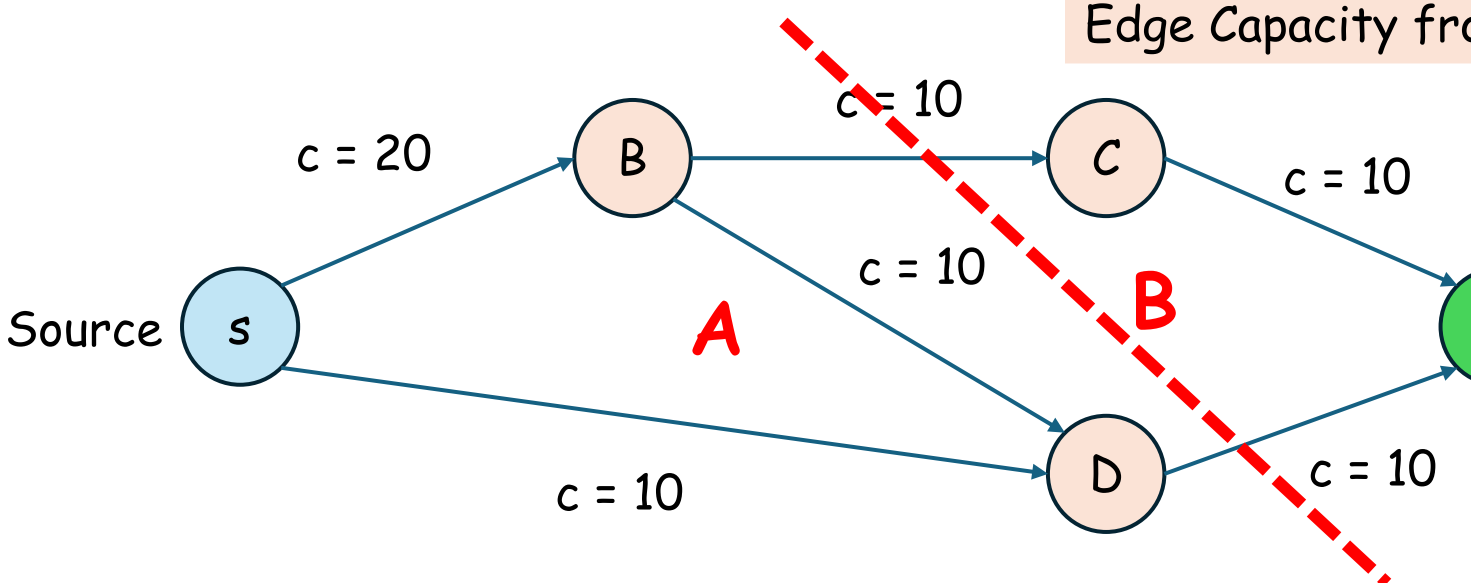
Edge Capacity from A to B = 10 + 10 + 10 = 30



upper bounds on an s-t flow

- A basic obstacle to the existence of large flows is as follows: Suppose we divide the nodes of the graph into two sets, A and B, so that s ∈ A and t ∈ B. Then, intuitively, any flow that goes from s to t must cross from A into B at some point, and thereby use up some of the edge capacity from A to B. This suggests that each such division (cut) of the graph puts a bound on the maximum possible flow value.
- We will learn later that the maximum-flow value equals the minimum capacity of any such division, called the minimum cut.

Edge Capacity from A to B = 10 + 10 = 20



Minimum cut!

Max-Flow Problem?

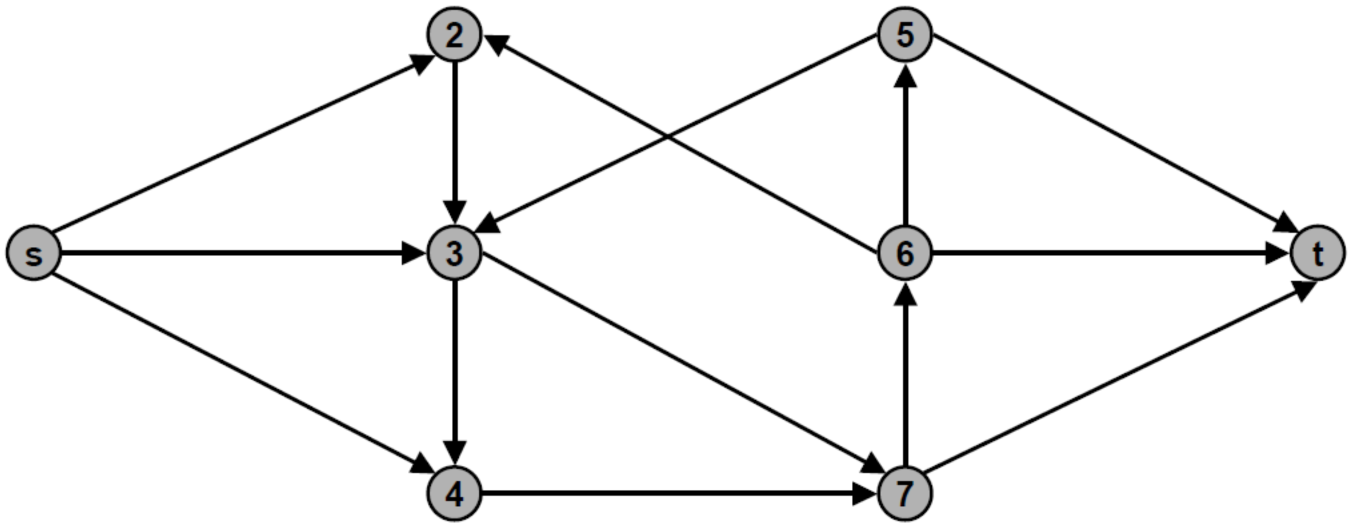
Is this an “abstract hypothetical problem” or what?

What are the applications of Max-Flow Problem?

Application of "Max-Flow" for **Communication Networks**

Disjoint path problem:

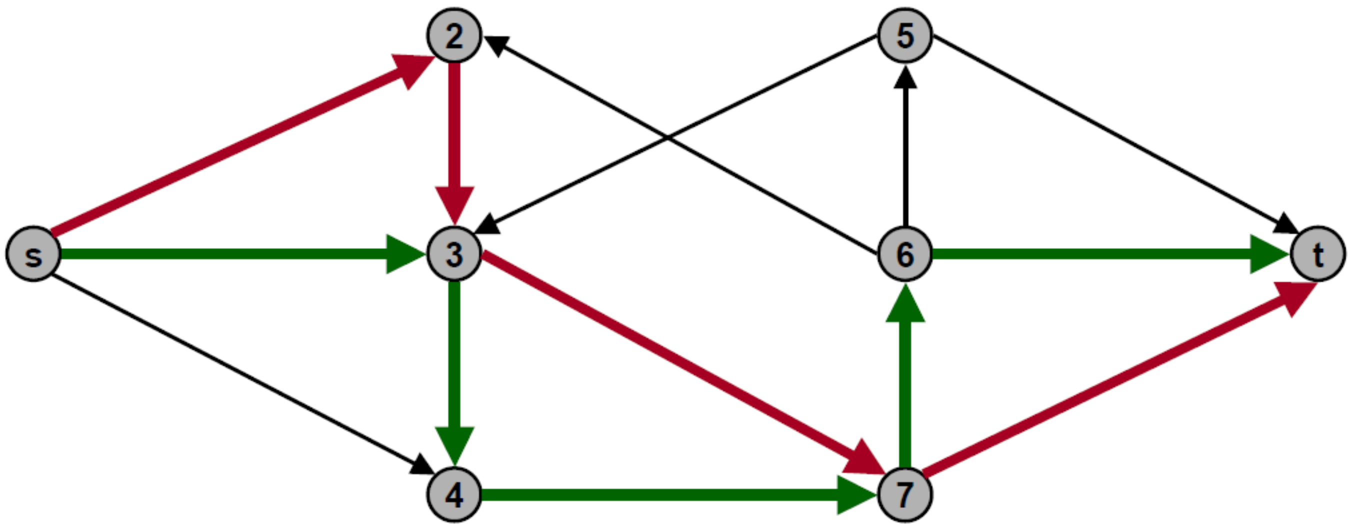
find max number edge-disjoint s-t path



Application of "Max-Flow" for **Communication Networks**

Disjoint path problem:

find max number edge-disjoint s-t path

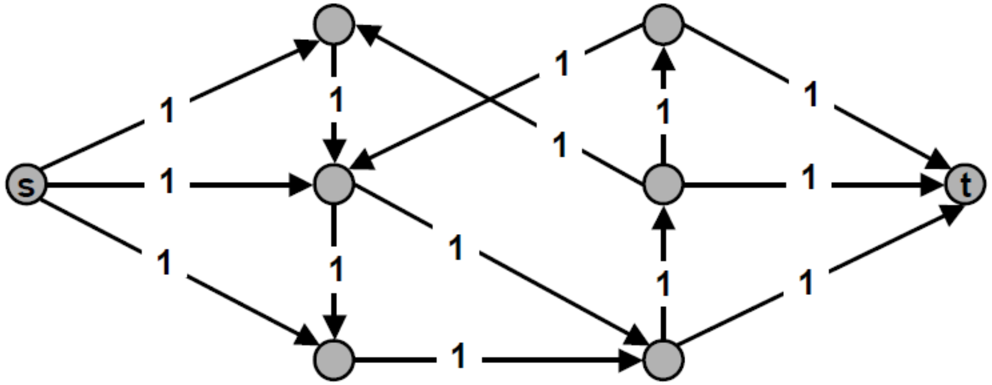


Application of "Max-Flow" for **Communication Networks**

Disjoint path problem:

find max number edge-disjoint s-t path

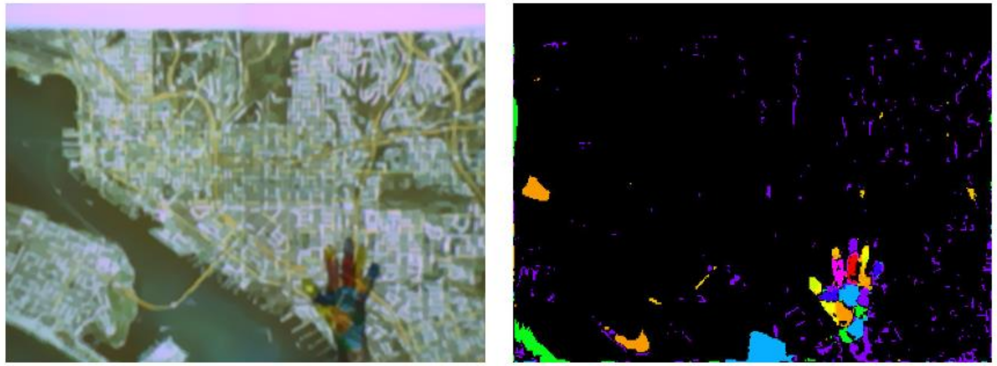
Max-Flow Formulation: assign unit capacity to every edge



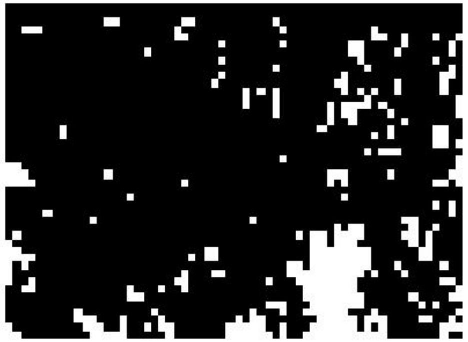
Theorem: There are k edge-disjoint s-t paths if and only if the max flow value is k.
We will study this problem later on ...

Application of "Max-Flow" for (**Binary**) **Image Segmentation**

Example: Given a binary image, the purpose of the segmentation is to track **the position of the hand** in camera images for gestural interaction.



(a) (b)

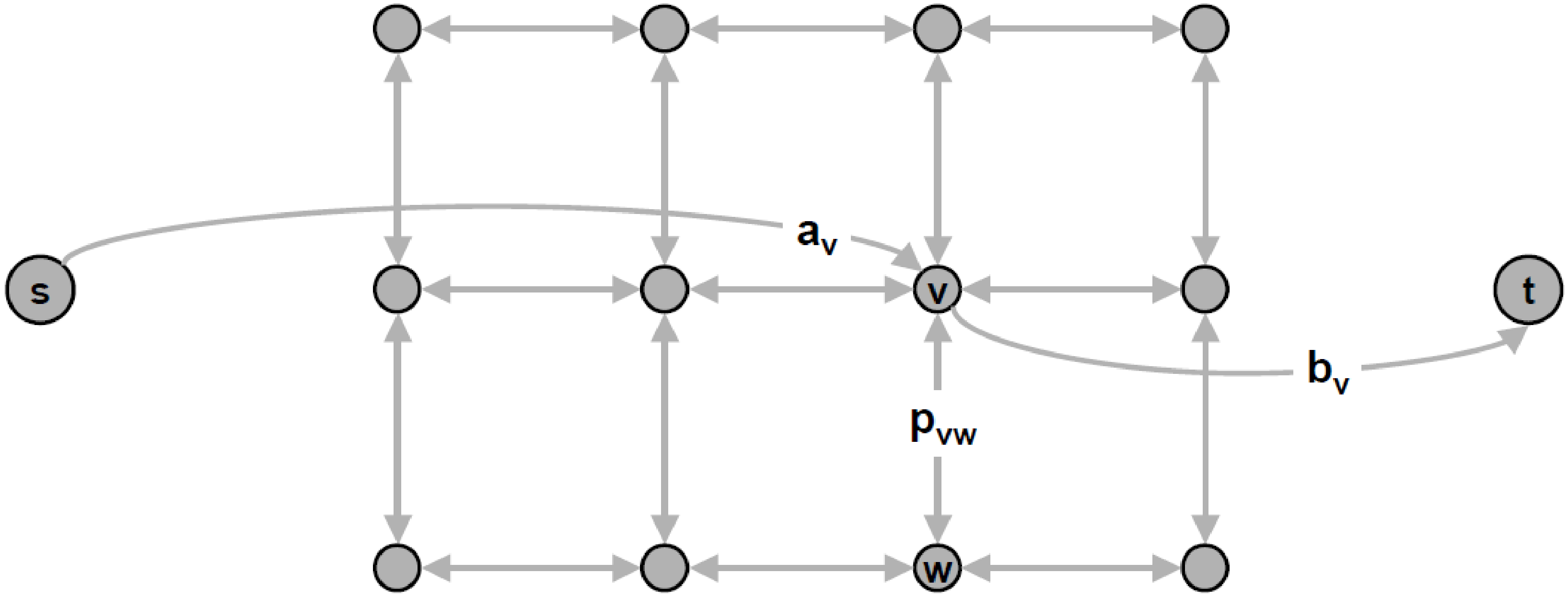


(c)

http://people.csail.mit.edu/yingyin/resources/doc/projects/yingyin_6854_project.pdf

Application of “Max-Flow” for (**Binary**) **Image Segmentation**

Max-Flow Formulation:
a_v = likelihood pixel v in foreground
b_v = likelihood pixel v in background
P_{vw} = separation penalty for labeling one of v and w as foreground, and the other as background



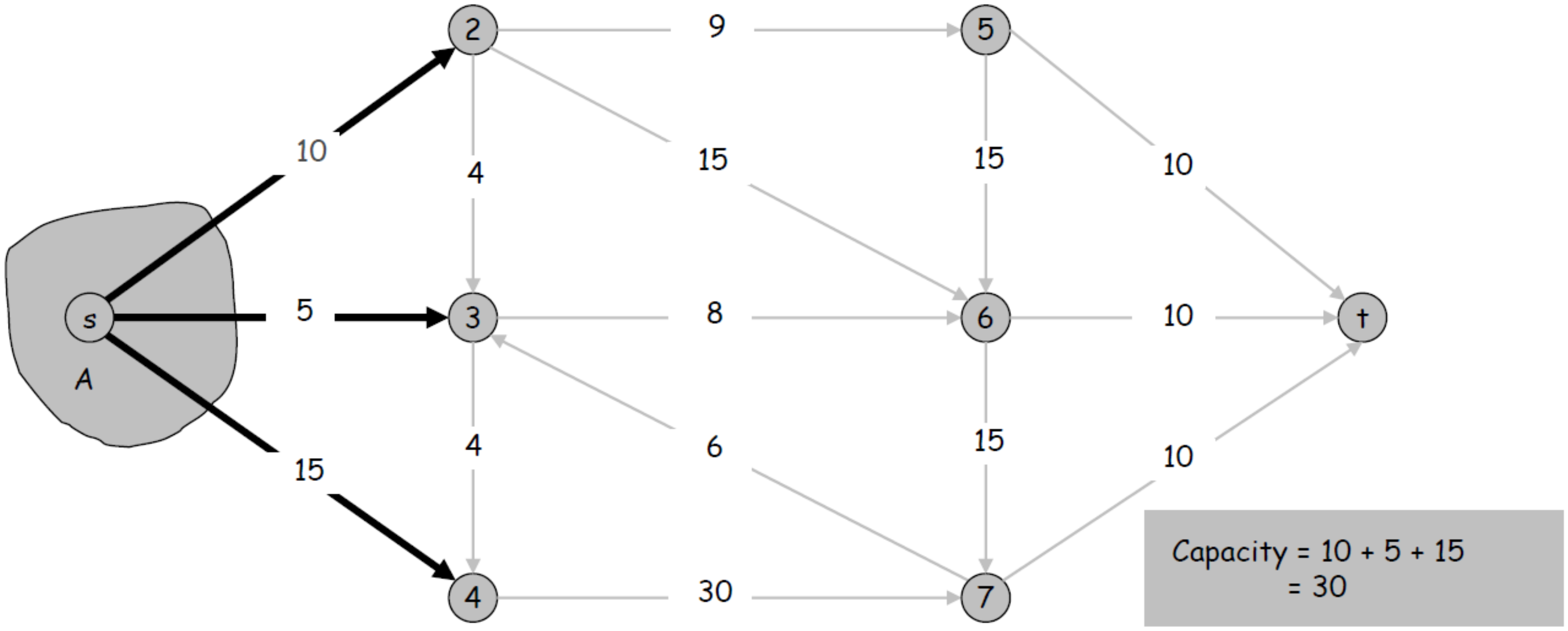
Algorithm for Finding Max-Flow?

Just be patient… before we learn an algorithm, we must understand basic terminologies and definitions

Flow Network and Cuts

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

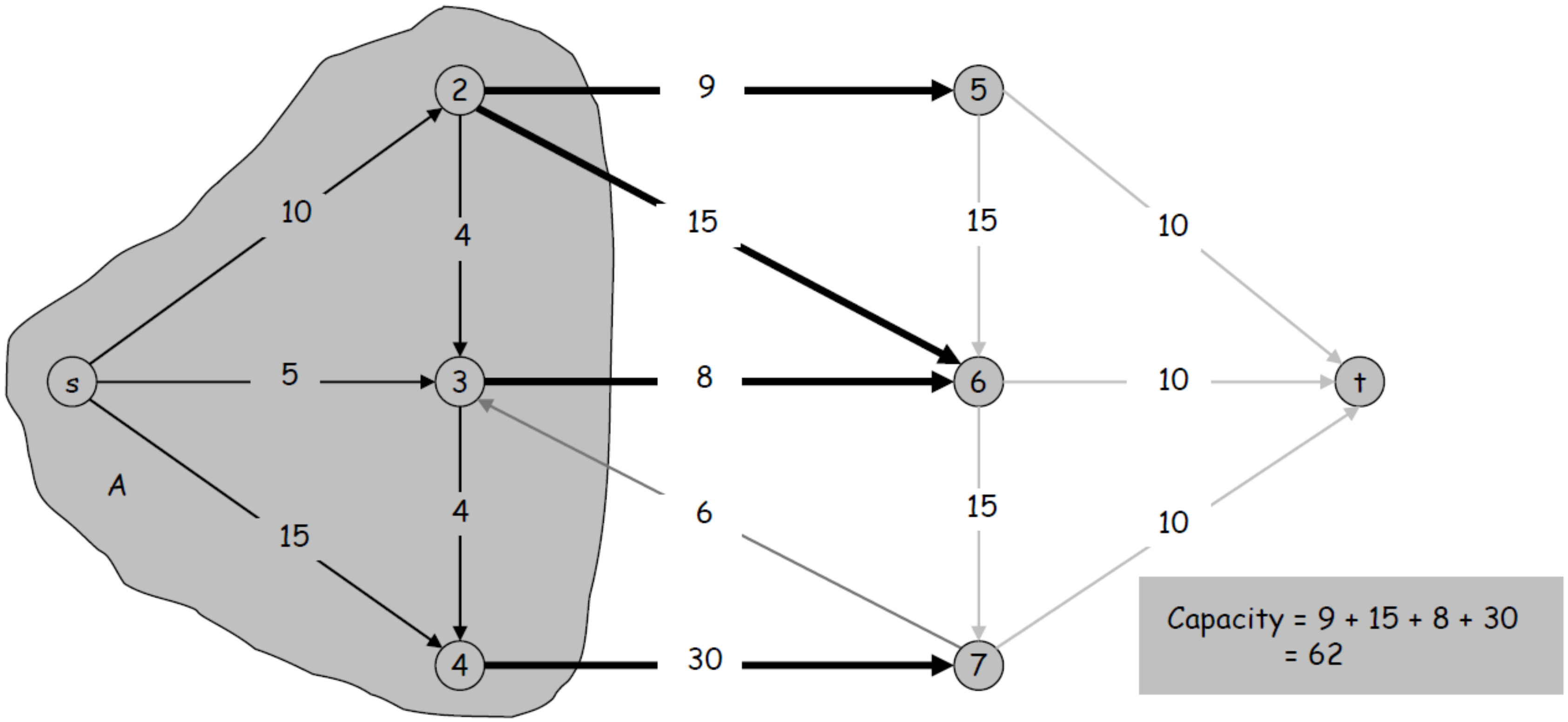
Definition. The capacity of a cut (A, B) is: $$cap(A,B) = \sum_{e \text{ out of } A} c(e)$$



Flow Network and Cuts

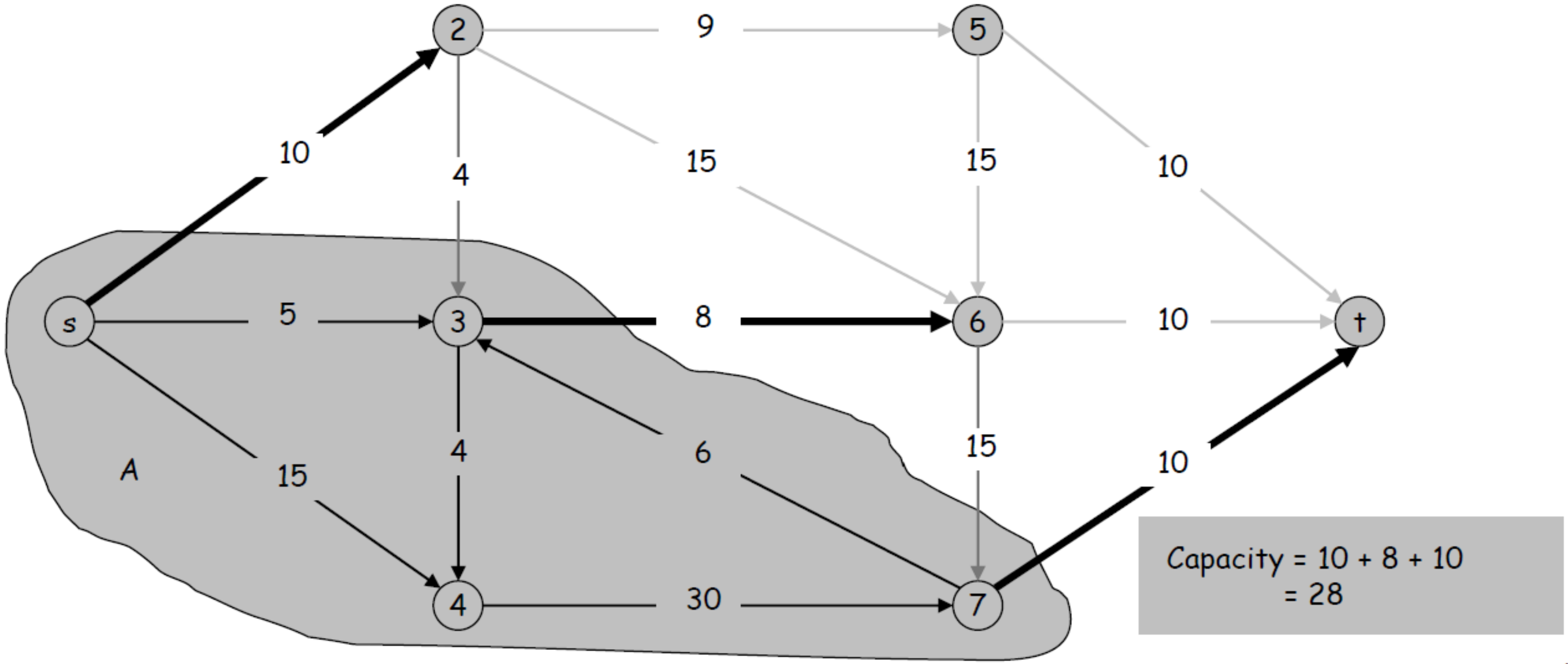
Definition. An s-t cut is a partition (A, B) of V with s ∈ A and t ∈ B.

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



Minimum Cut Problem

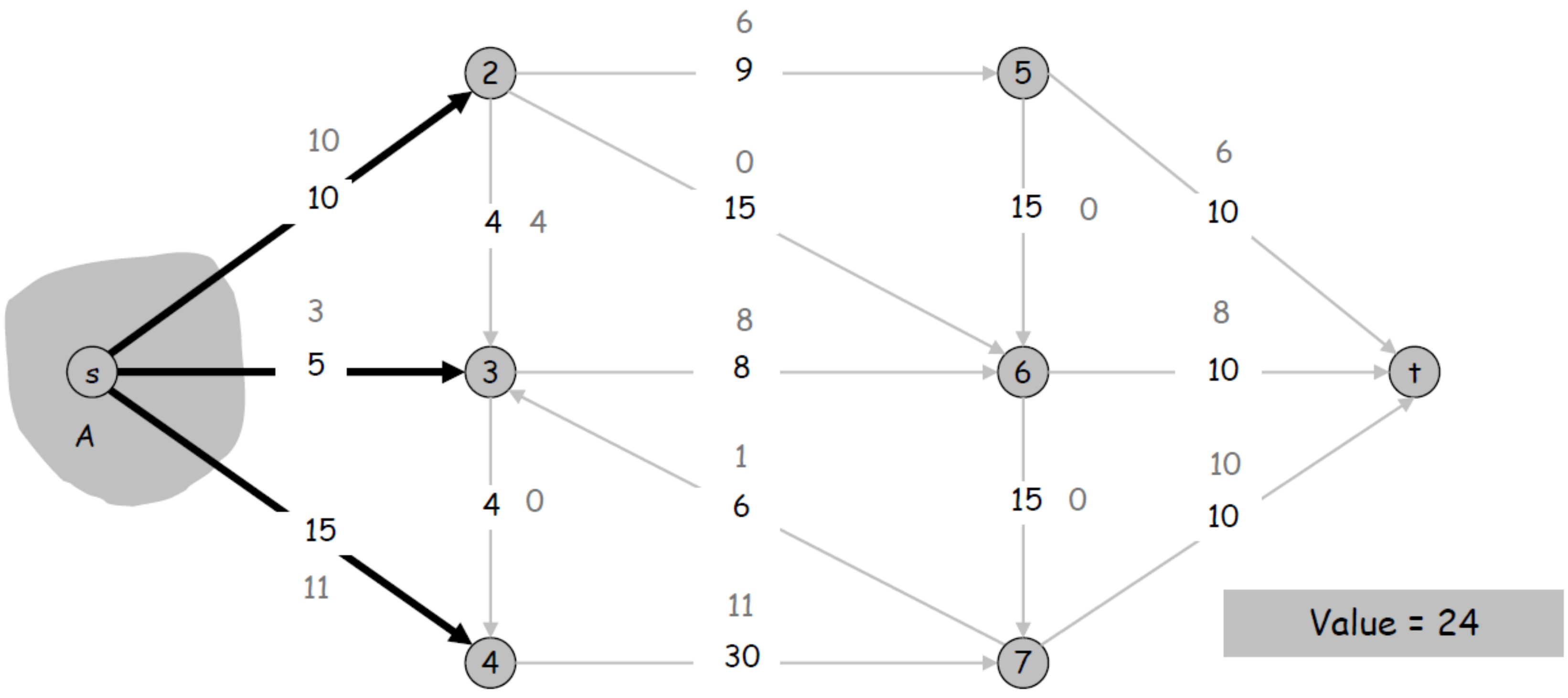
Min s-t cut problem. Find an s-t cut of minimum capacity.



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.

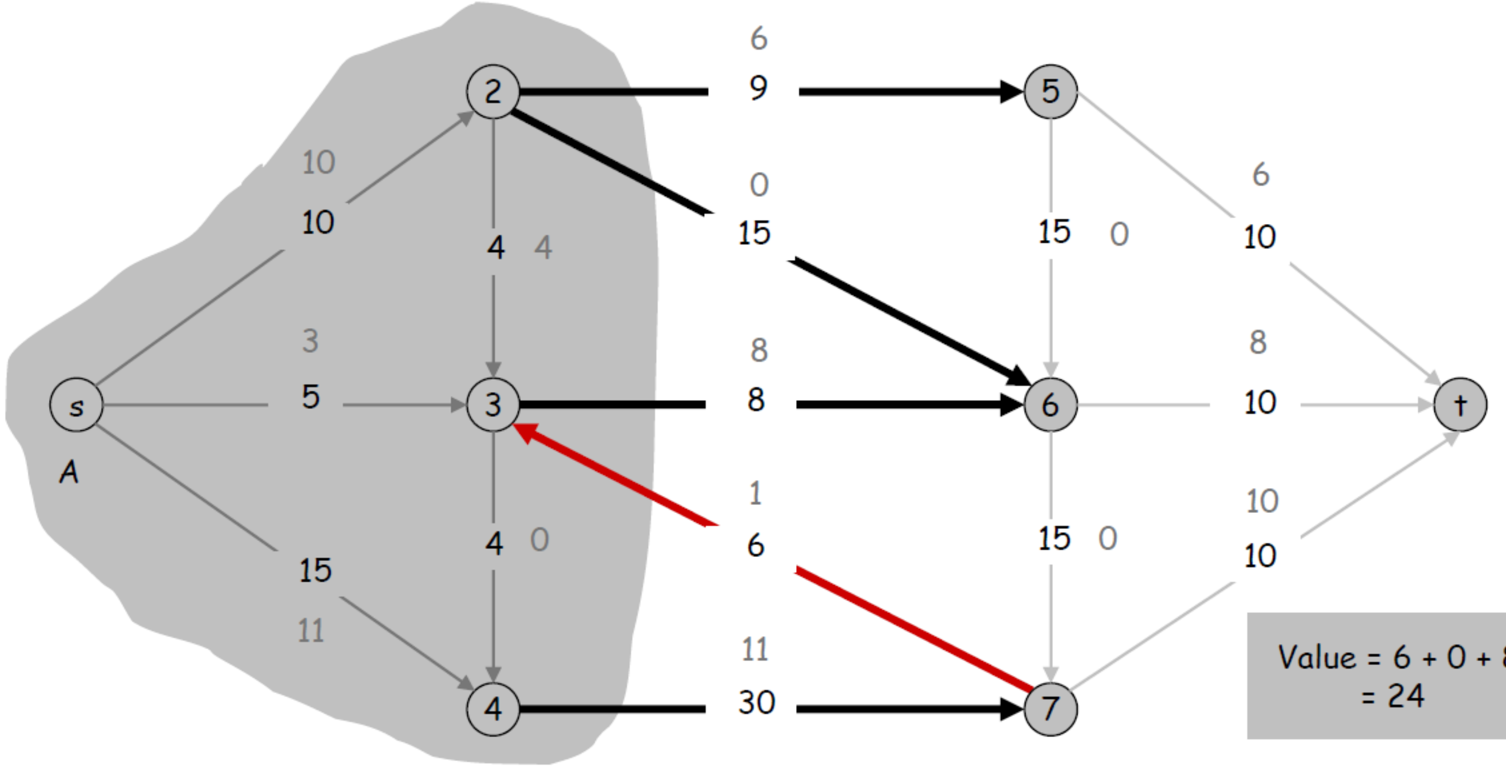
$$\sum_{e \text{ out of } A} f(e) \;-\; \sum_{e \text{ in to } A} f(e) \;=\; v(f)$$



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.

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

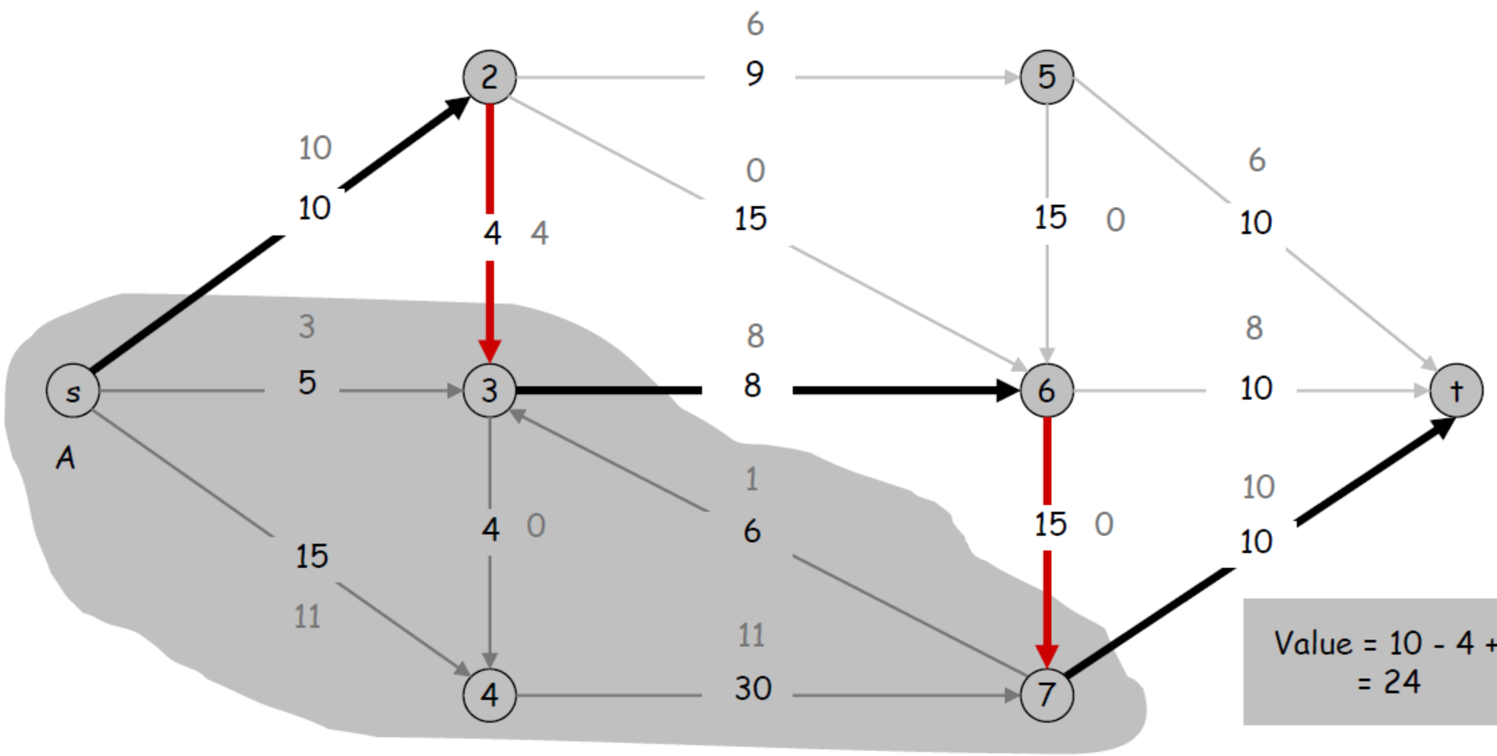


Value = 6 + 0 + 8 - 1 + 11
= 24

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.

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



Value = 10 - 4 + 8 - 0 + 10
= 24

Flows and Cuts

Flow value lemma. Let f be any flow, and let (A, B) be any s-t cut. Then

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

Proof.

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

by flow conservation, all terms except v = s are 0 $\longrightarrow$

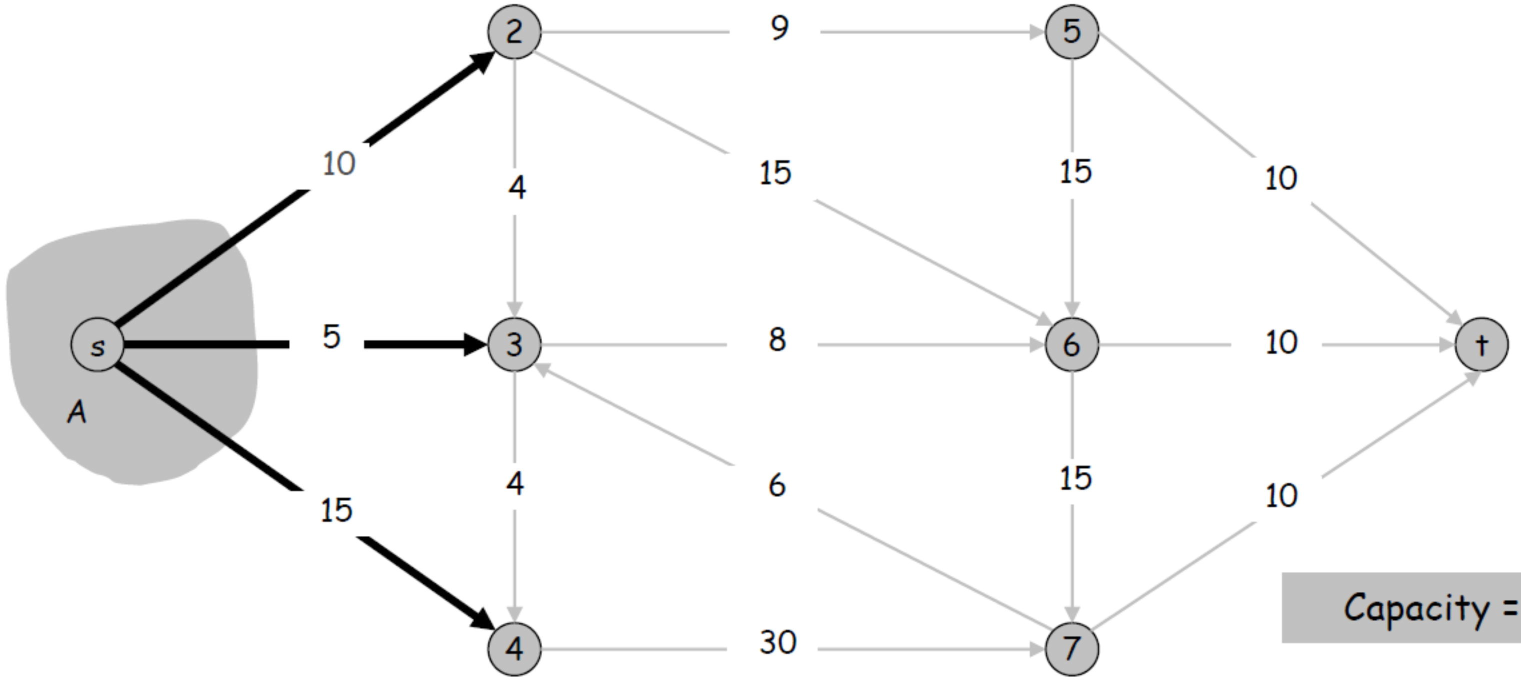
$$= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to v}} f(e) \right)$$

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

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



Capacity = 30

Flows and Cuts

Weak duality. Let f be any flow. Then, for any s-t cut (A, B) we have $v(f) \le \text{cap}(A, B)$.

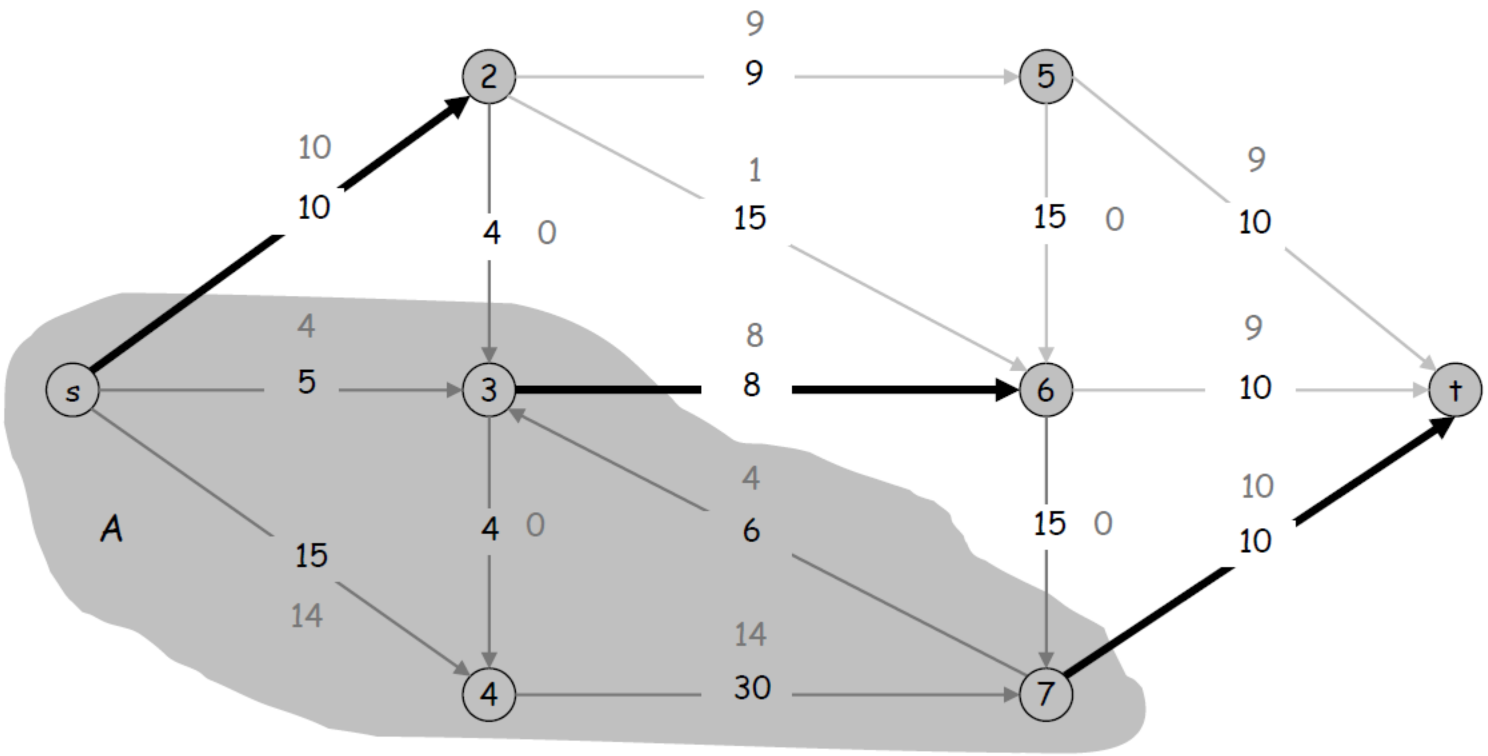
Proof.

$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\le \sum_{e \text{ out of } A} f(e) \\
&\le \sum_{e \text{ out of } A} c(e) \\
&= \text{cap}(A, B) \quad \blacksquare
\end{aligned}
$$

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



OK, Let's Discuss an Algorithm

First, it is necessary to understand what "Residual Graph" is

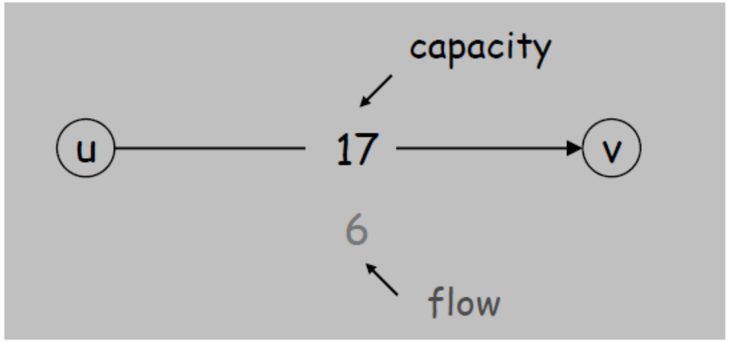
Residual Graph

Given a flow network G, and a flow $\mathbf{f}$ on G, we define the **residual graph** G_f of G with respect to $\mathbf{f}$ as follows:

- The node set of G_f is the same as that of G;
- For each edge $\mathbf{e}$ of G on which $\mathbf{f(e) < c(e)}$, there are $\mathbf{c(e) - f(e)}$ "leftover" units of capacity on which we can try pushing flow forward. These are **forward edges**.
- For each edge $\mathbf{e}$ of G on which $\mathbf{f(e) > 0}$, there are $\mathbf{f(e)}$ units of flow that we can "undo" if we want to, by pushing flow **backward**. So we include the edge $\mathbf{e}^R$ in G_f, with a capacity of $\mathbf{f(e)}$.

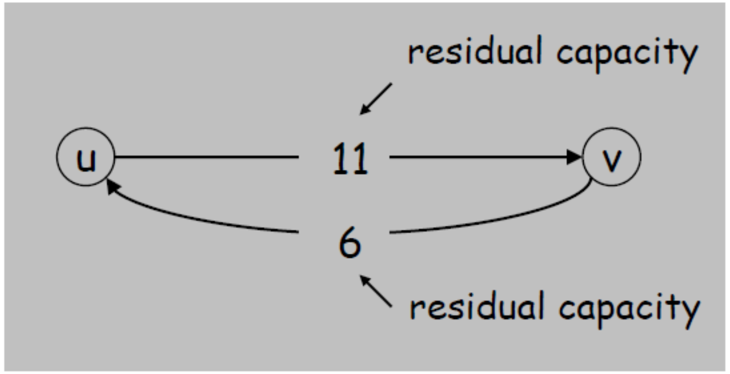
Residual Graph

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:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{for forward edge} \\ f(e) & \text{for backward edge} \end{cases}$$



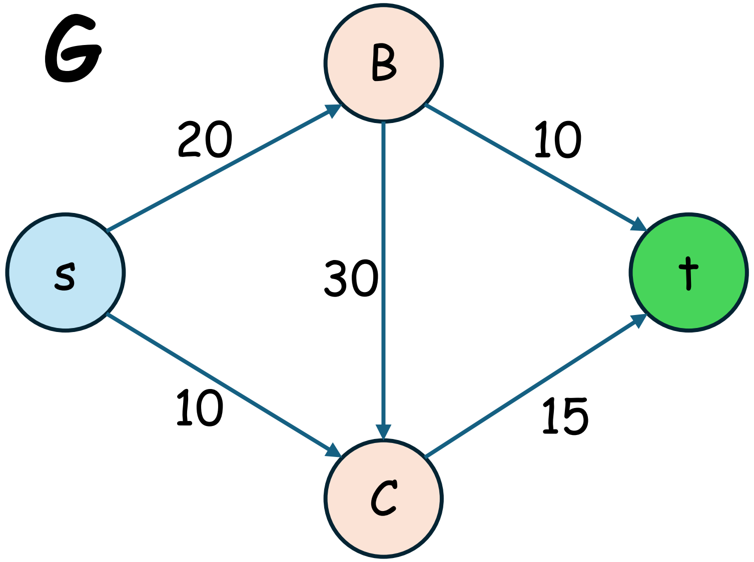
Residual graph: $G_f = (V, E_f)$.
- Residual edges with positive residual capacity.
- $E_f = \{e\} \cup \{e^R\}$.

Residual Capacity

- Note that each edge **e** in ***G*** can give rise to one or two edges in G_f. If $0 < f(e) < c(e)$ it results in both a forward edge and a backward edge being included in G_f;
  - Thus, G_f has at most twice as many edges as ***G***
- We will refer to the capacity of an edge in the residual graph as a **residual capacity**.

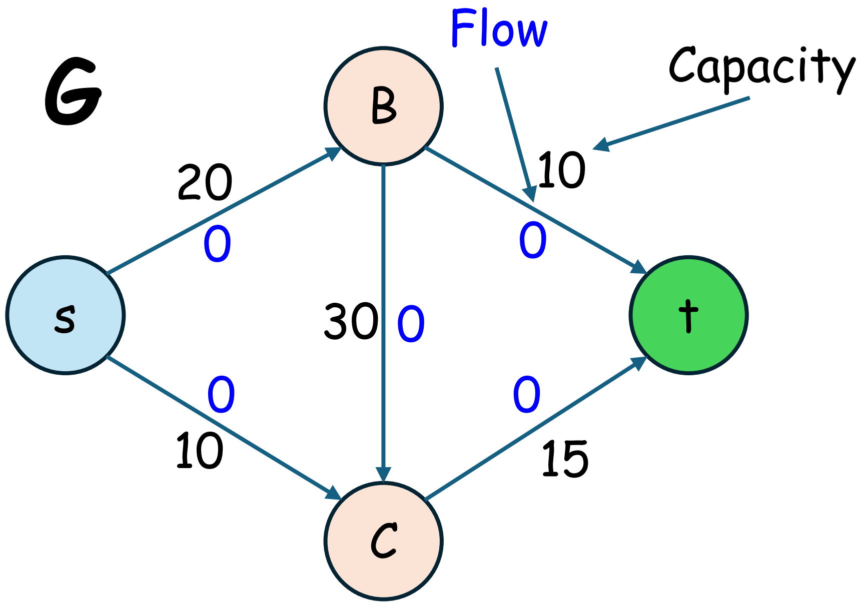
What is "Residual Graph" for?

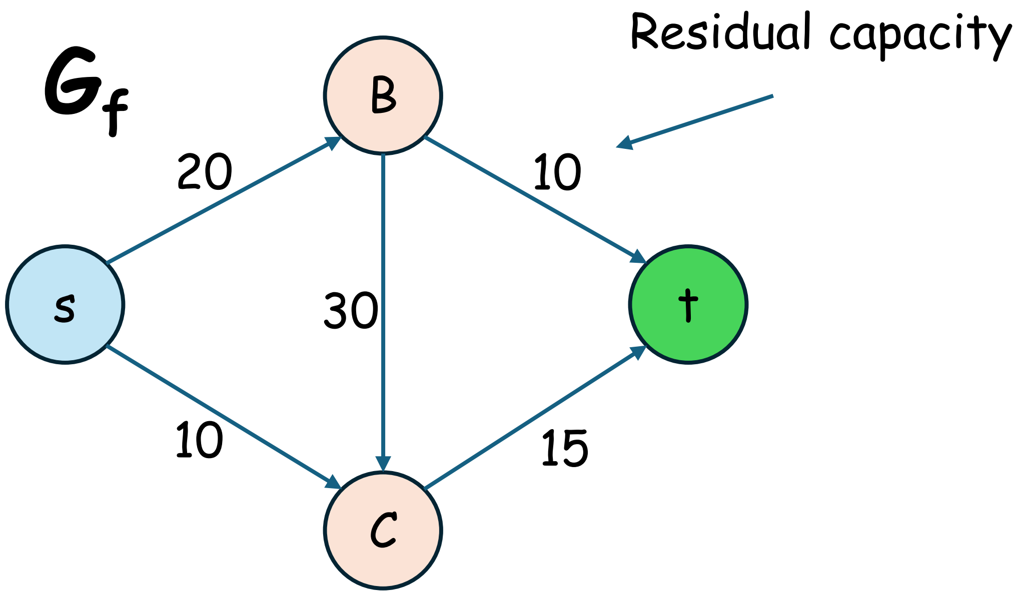
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in G;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on P;
  - Augment flow along path P --> set a new flow in G and update G_f;
  - **Repeat** until no new path P is found.



What is "Residual Graph" for?

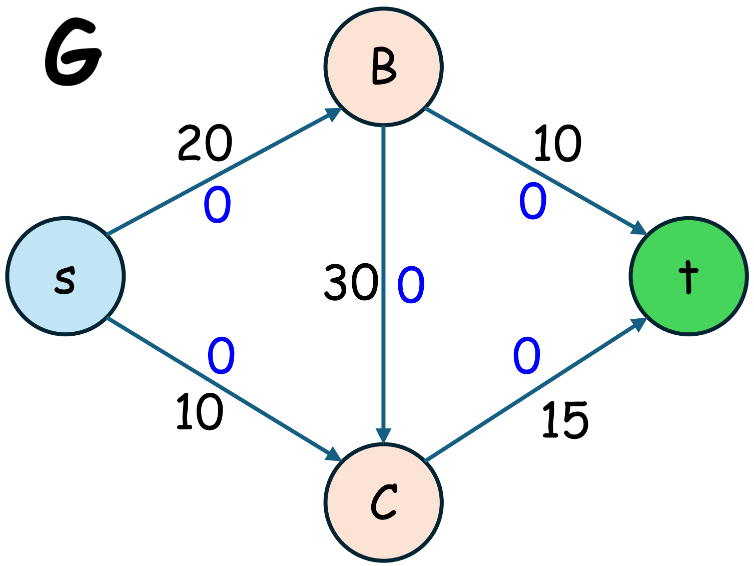
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

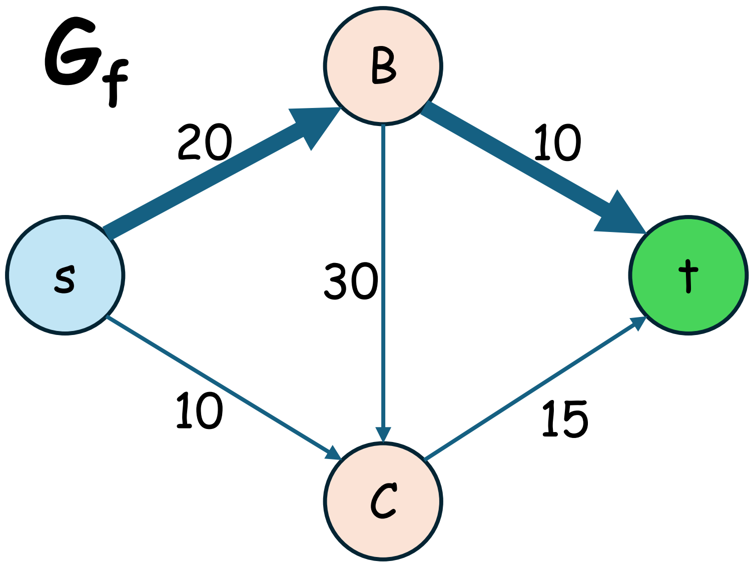




What is "Residual Graph" for?

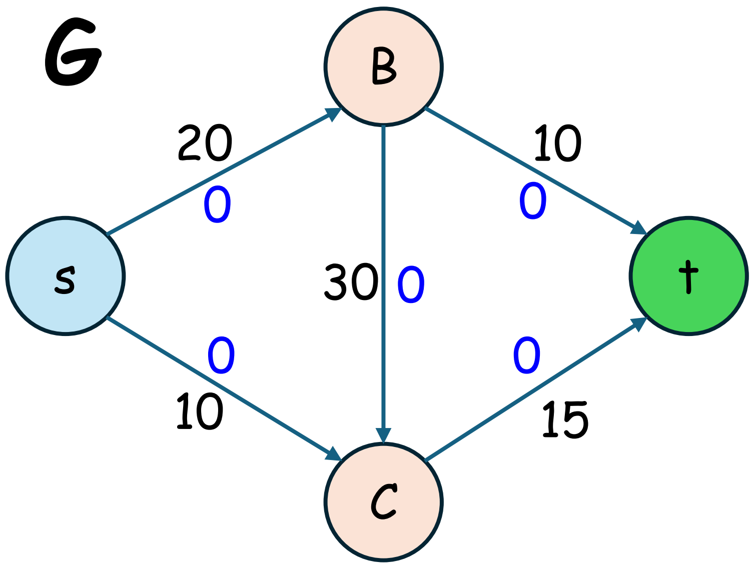
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

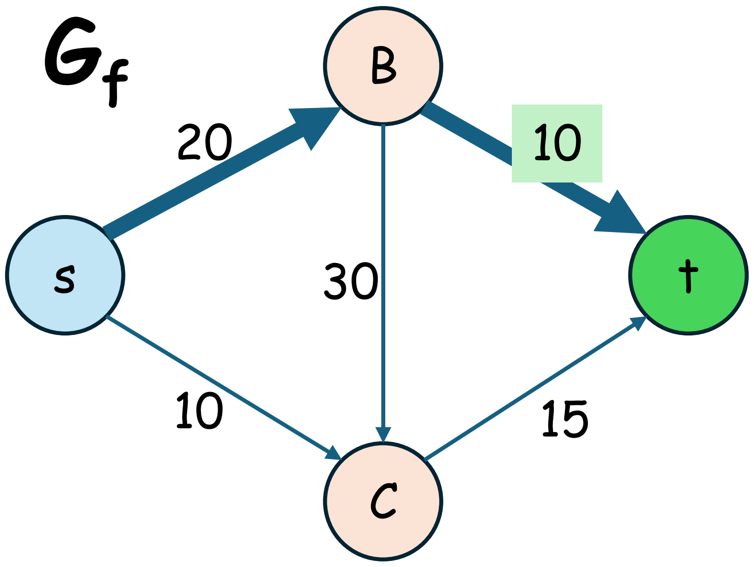




What is "Residual Graph" for?

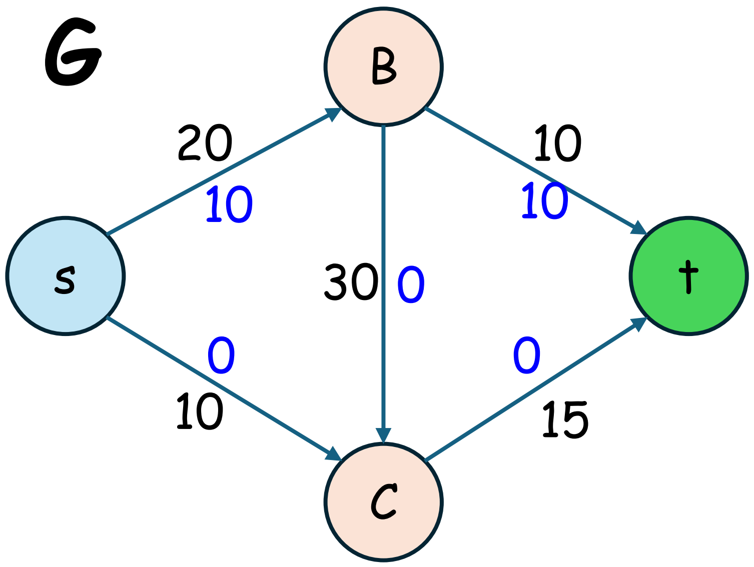
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in *G*;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on P;
  - Augment flow along path P --> set a new flow in *G* and update G_f;
  - **Repeat** until no new path P is found.

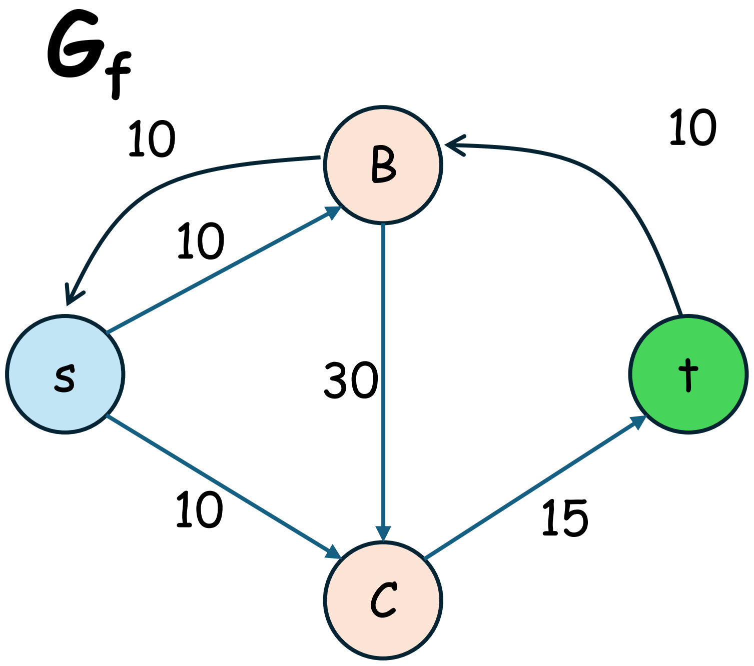




What is "Residual Graph" for?

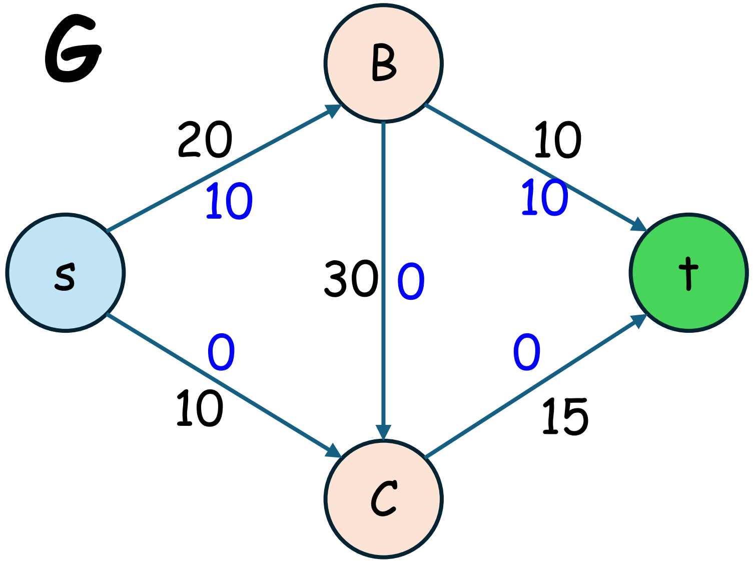
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

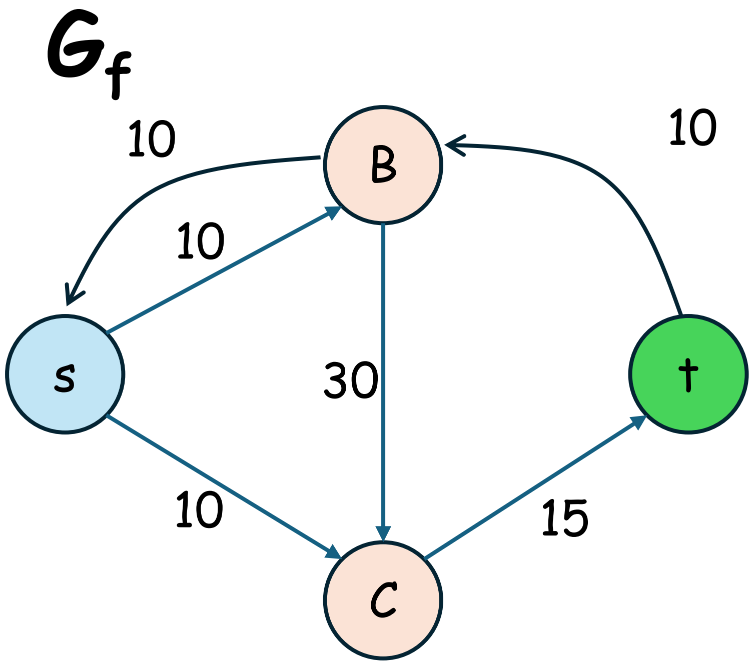




What is "Residual Graph" for?

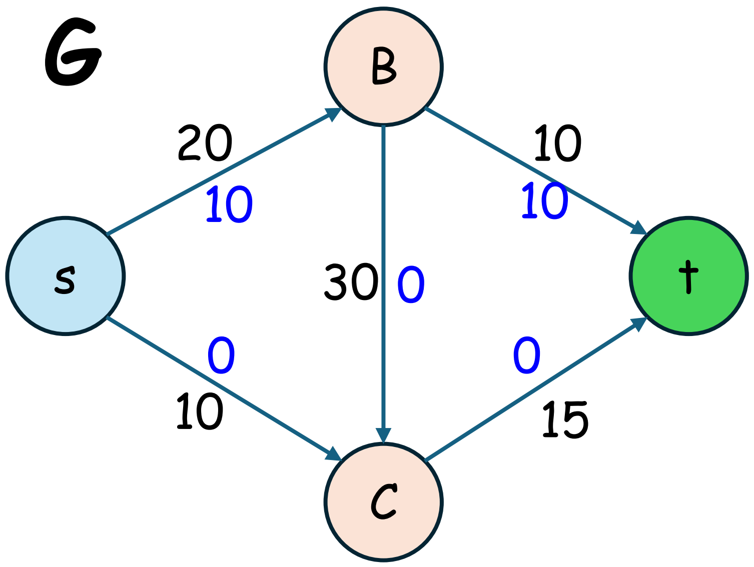
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

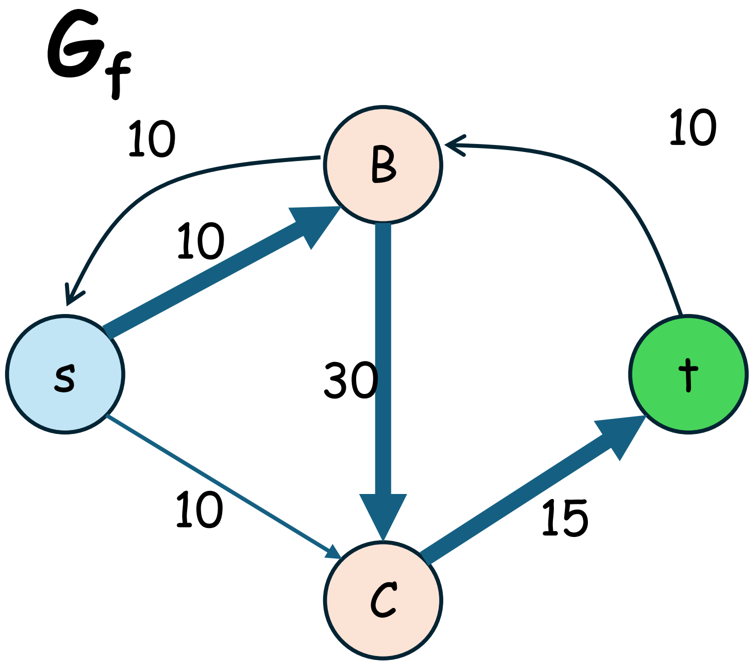




What is "Residual Graph" for?

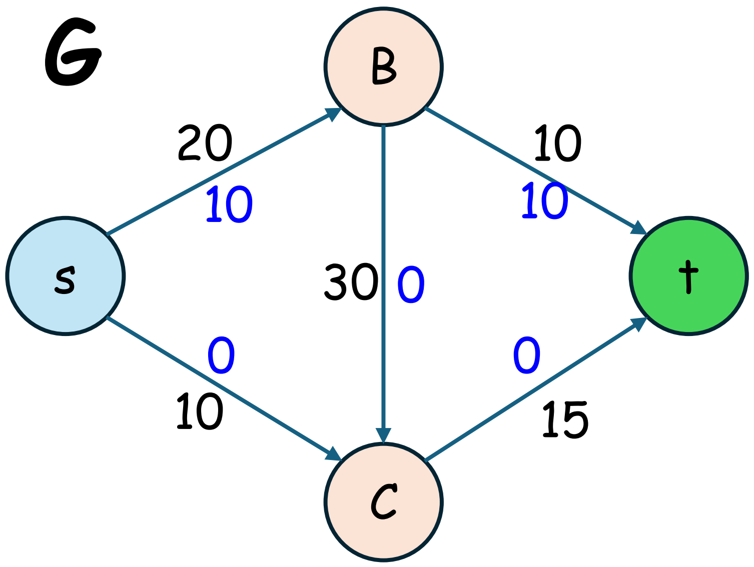
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

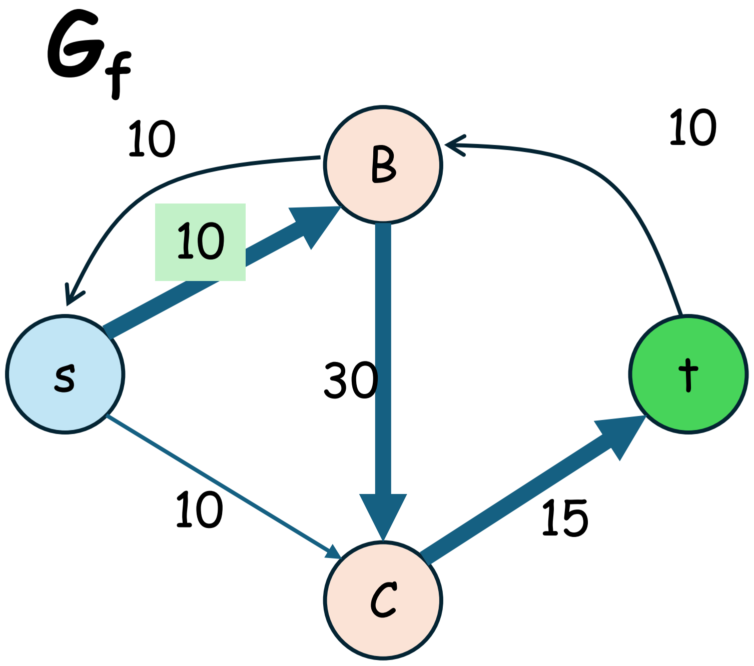




What is "Residual Graph" for?

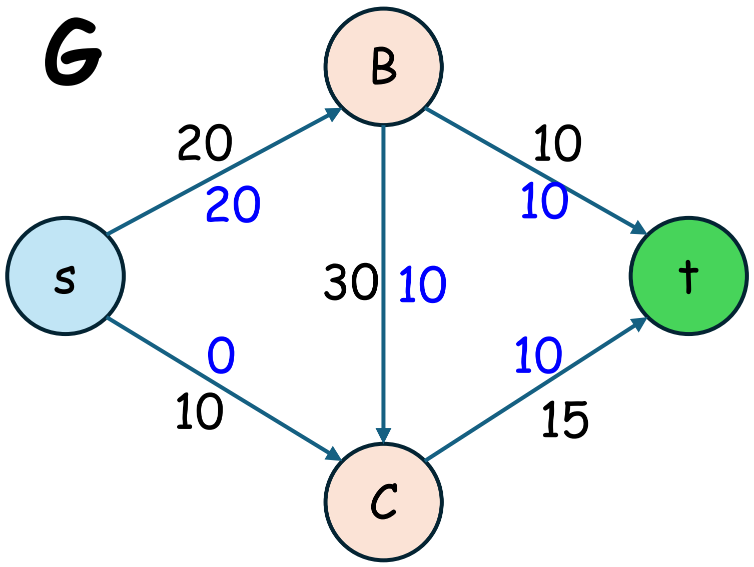
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in G;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on P;
  - Augment flow along path P --> set a new flow in G and update G_f;
  - **Repeat** until no new path P is found.

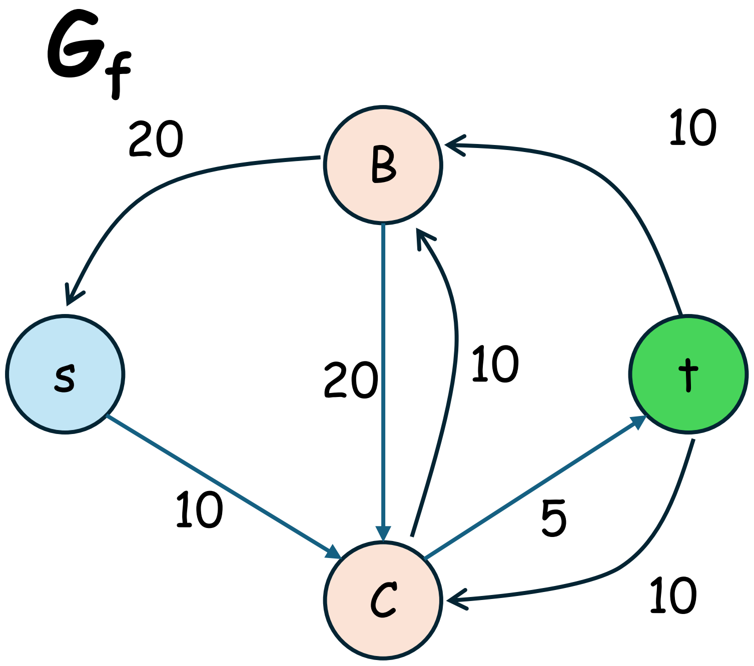




What is "Residual Graph" for?

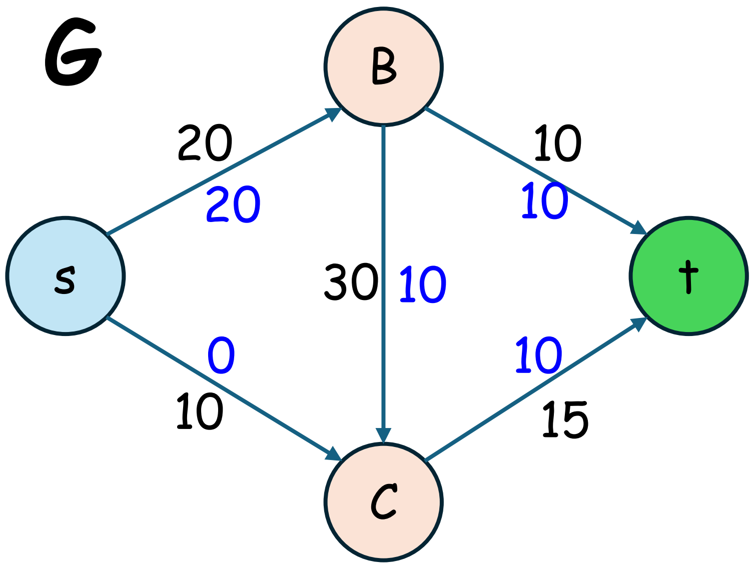
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

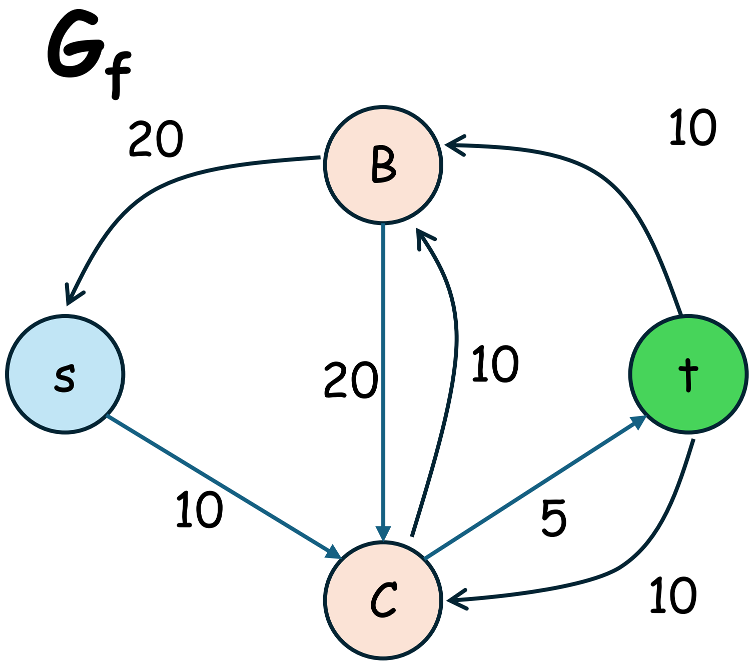




What is "Residual Graph" for?

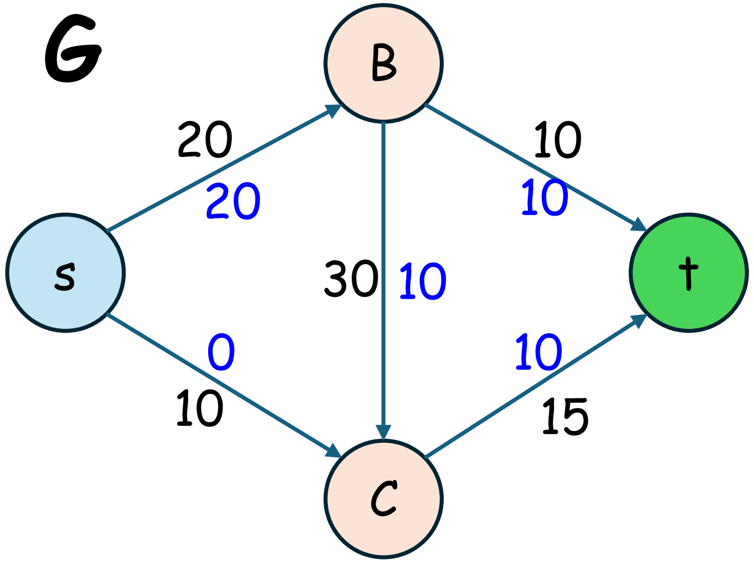
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

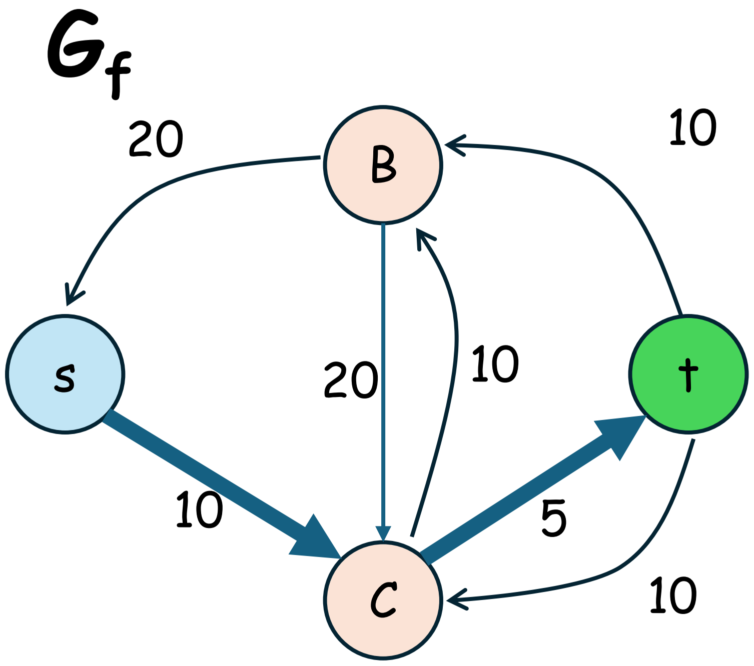




What is "Residual Graph" for?

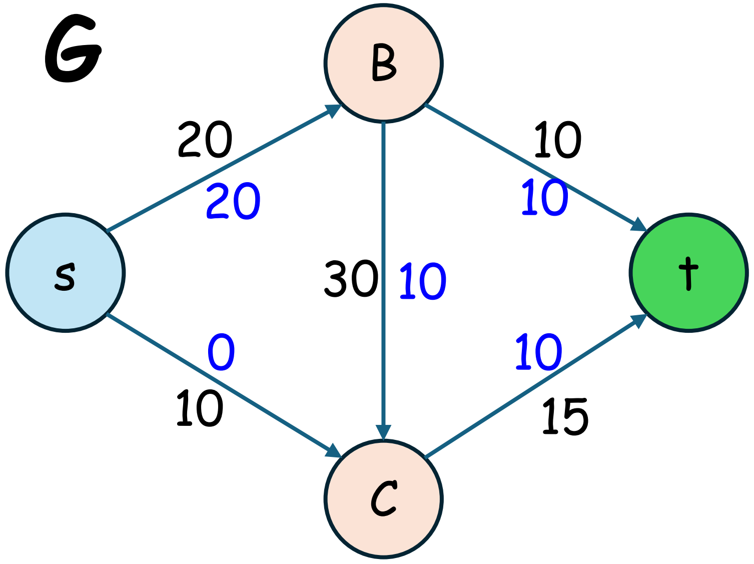
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in ***G***;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in ***G*** and update G_f;
  - **Repeat** until no new path **P** is found.

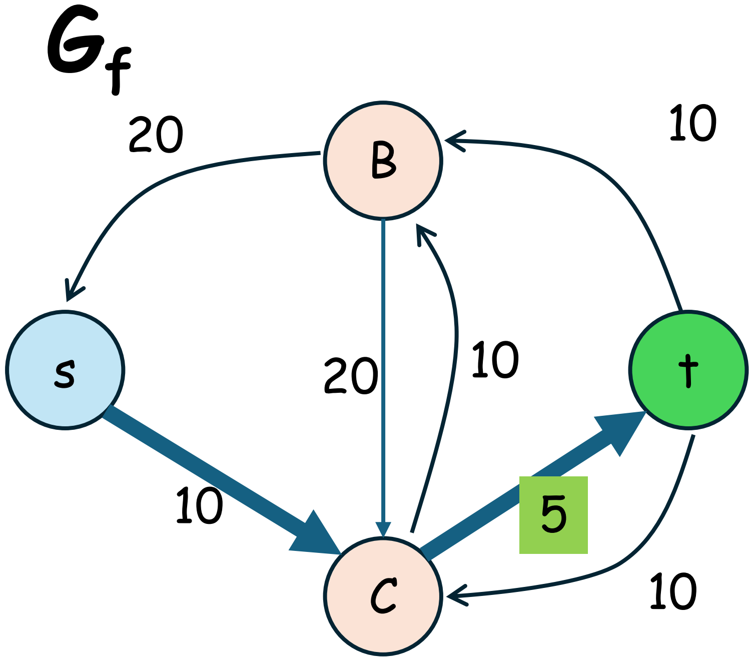




What is "Residual Graph" for?

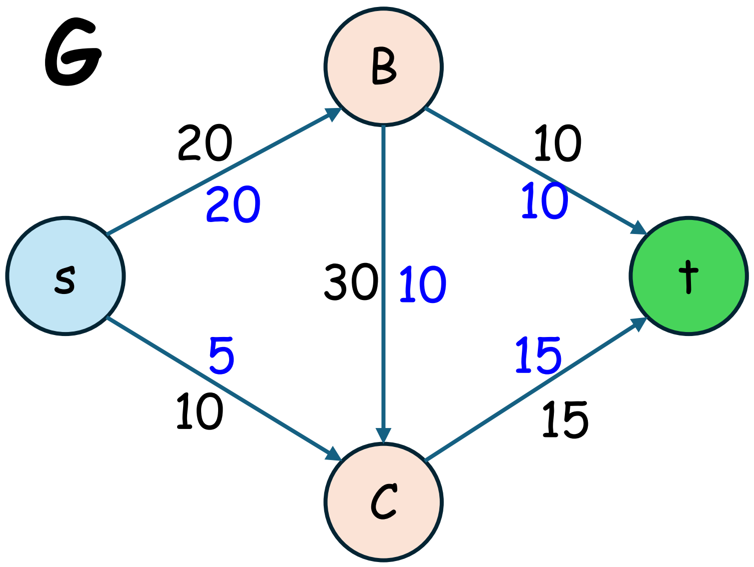
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in G;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on P;
  - Augment flow along path P --> set a new flow in G and update G_f;
  - **Repeat** until no new path P is found.

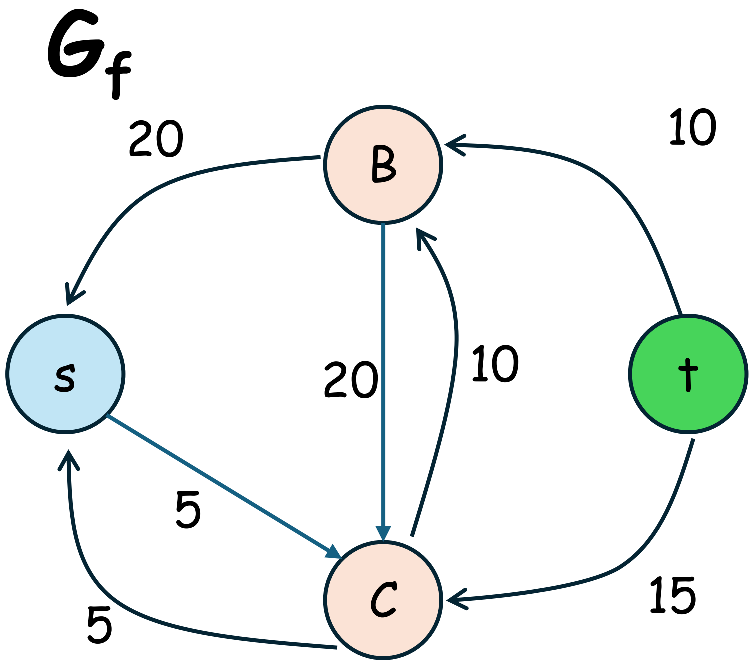




What is "Residual Graph" for?

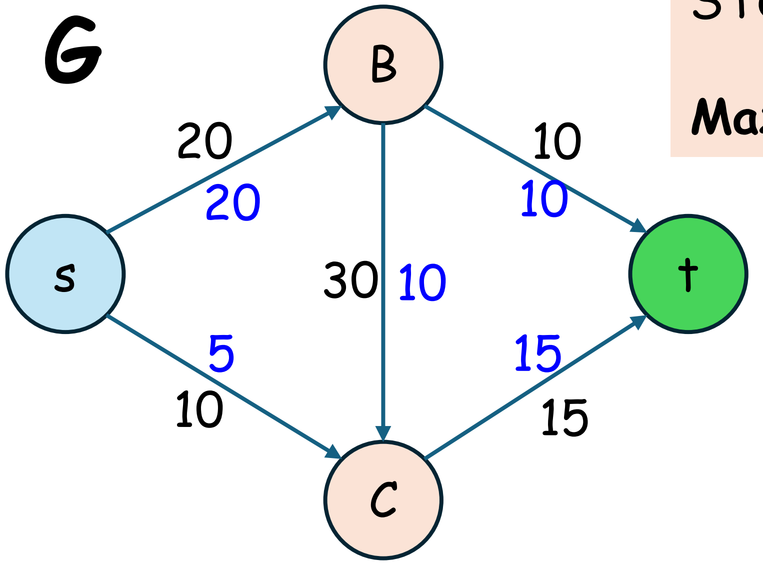
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in G;
  - Find s-t path P on G_f;
  - Find the minimum **residual capacity** of any edge on P;
  - Augment flow along path P --> set a new flow in G and update G_f;
  - **Repeat** until no new path P is found.





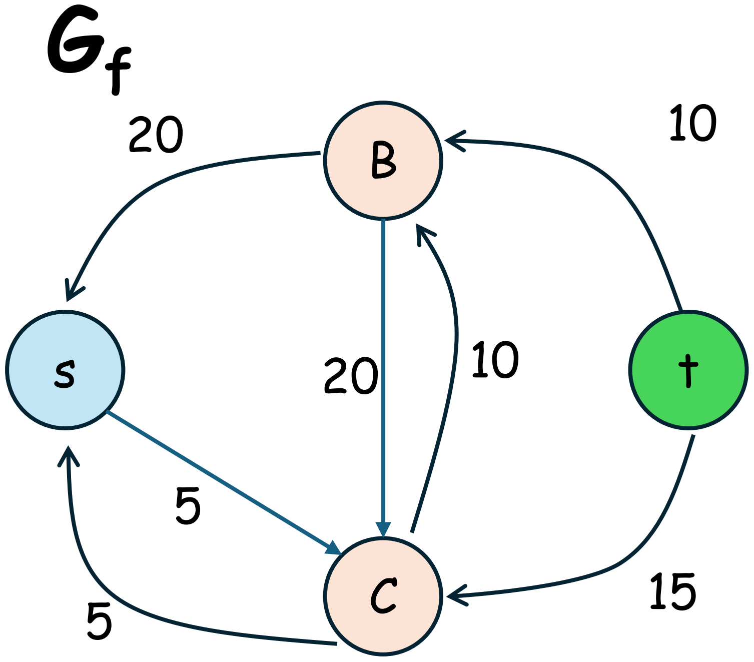
What is "Residual Graph" for?

- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in G;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in G and update G_f;
  - **Repeat** until no new path **P** is found.



STOP!!

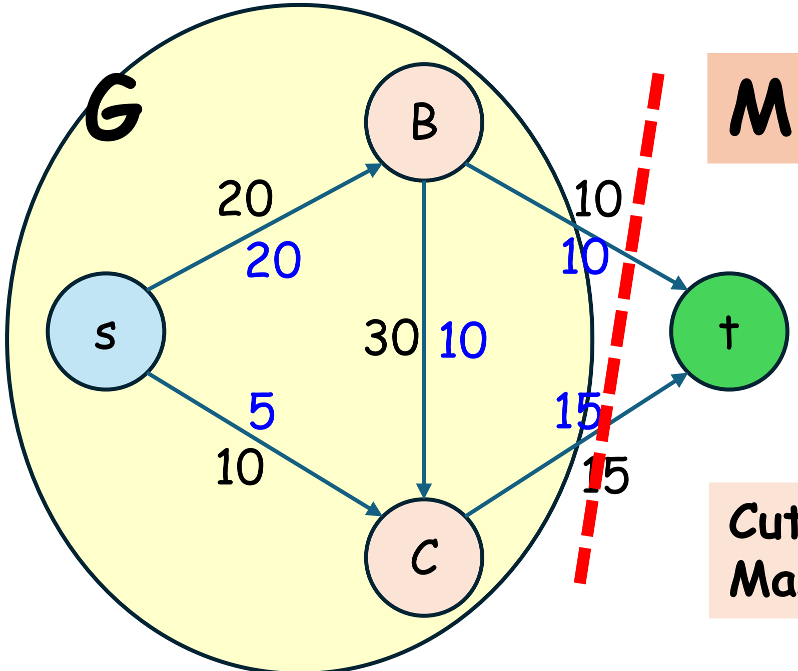
Max-Flow value = 25



What is "Residual Graph" for?

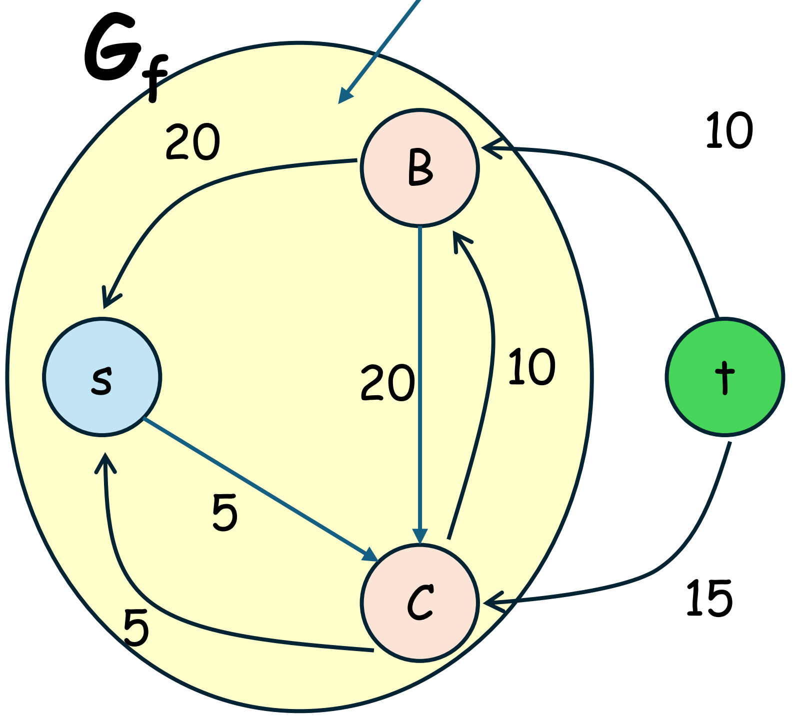
- Algorithm for finding Max-Flow
  - Start with **f(e) = 0** for all edge in **G**;
  - Find s-t path **P** on G_f;
  - Find the minimum **residual capacity** of any edge on **P**;
  - Augment flow along path **P** --> set a new flow in **G** and update G_f;
  - **Repeat** until no new path **P** is found.

All nodes reachable from s



Min-Cut

Cut Capacity = 25
Max-Flow = 25



Augmenting Paths in a Residual Graph

- Now we want to make precise the way in which we push flow from s to t in G_f.
- Let P be a simple s-t path in G_f— that is, P does not visit any node more than once.
- We define bottleneck(P, f) to be the minimum residual capacity of any edge on P, with respect to the flow f.
- Next we define the operation augment(f, P), which yields a new flow f' in G.
- It was purely to be able to perform this operation that we defined the residual graph.
- To reflect the importance of augment, one often refers to any s-t path in the residual graph as an augmenting path.
- This augmentation operation captures the type of forward and backward pushing of flow in the Ford-Fulkerson algorithm for computing an s-t flow in G.

Ford-Fulkerson Algorithm

Yes, you've just seen Ford-Fulkerson Algorithm!

```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
     f ← Augment(f, P)
     G_f ← Residual-Graf(G)    // update G_f

  return f
```

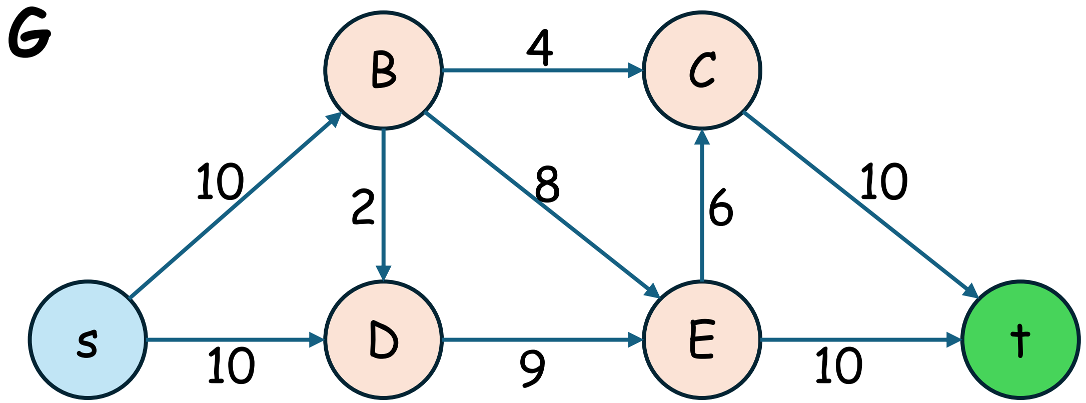
```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Forward edge

Reverse edge

```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
    f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
    f ← Augment(f, P)
    G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
    if e ∈ G.E then
      f(e) ← f(e) + b
    else
      f(e^R) ← f(e^R) - b

  return f
```

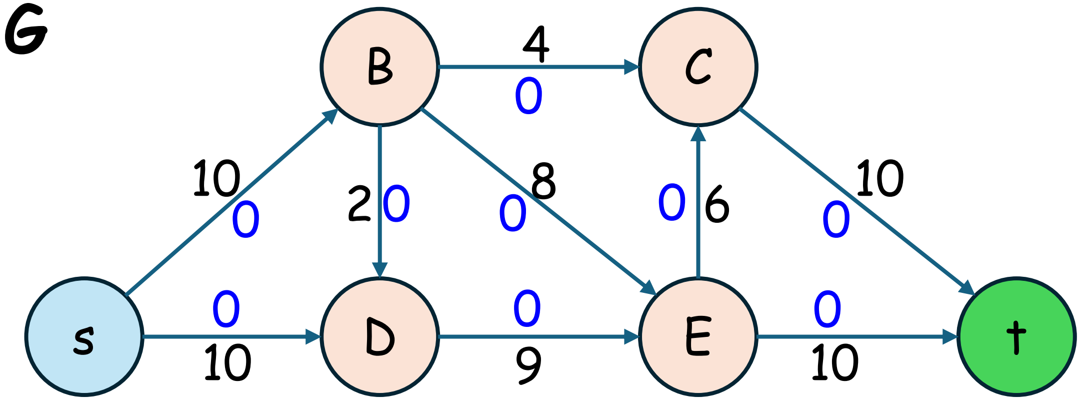


```
Fold-Fulkerson(G, s, t):
   for e ∈ G.E :
      f(e) ← 0

   G_f ← Residual-Graf(G)
   while there exist augmenting s-t path P on G_f:
      f ← Augment(f, P)
      G_f ← Residual-Graf(G)    // update G_f

   return f
```

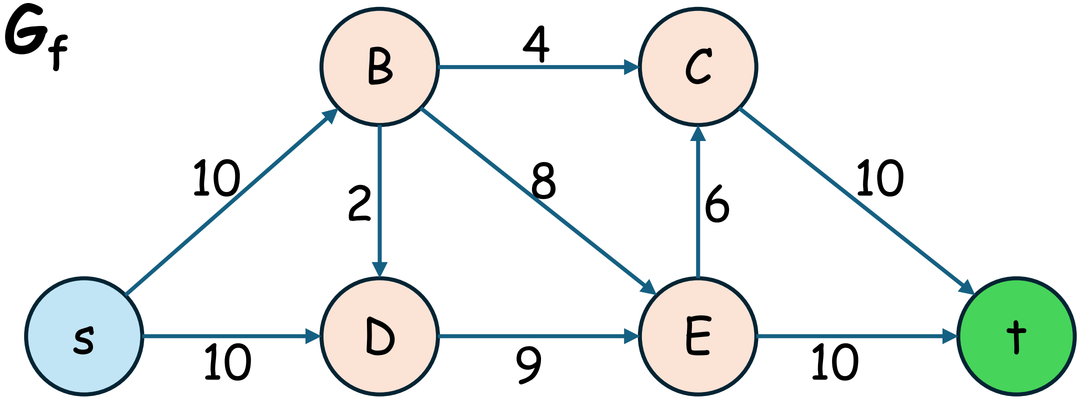
```
Augment(f, P):
   b ← Bottleneck(P, f)

   for e ∈ P:
      if e ∈ G.E then
         f(e) ← f(e) + b
      else
         f(e^R) ← f(e^R) - b

   return f
```

Flow value = 0

G



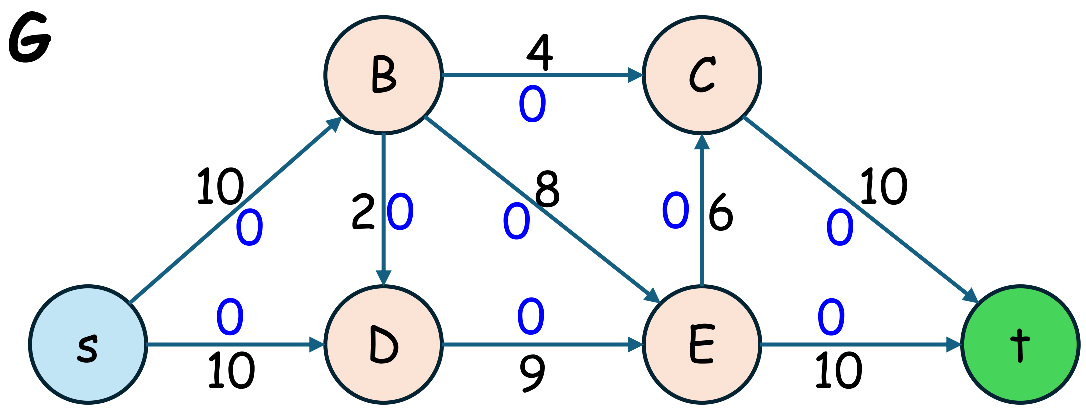
G_f

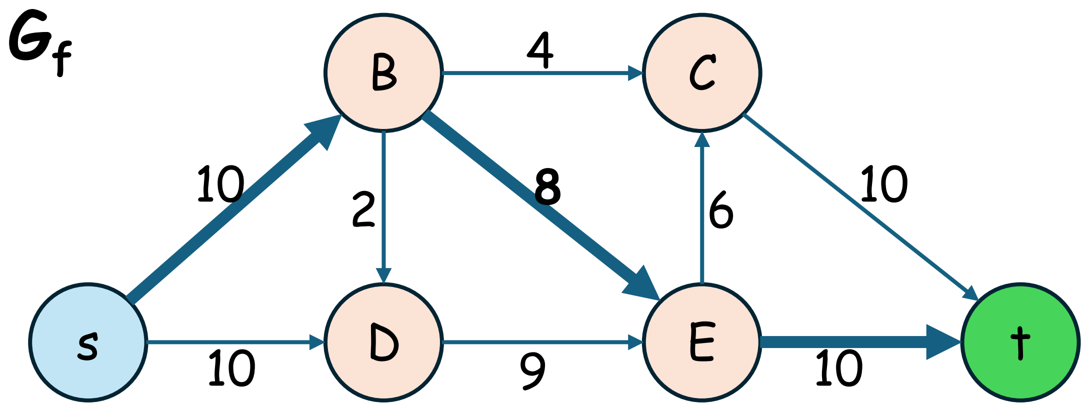


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
     f ← Augment(f, P)
     G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 0

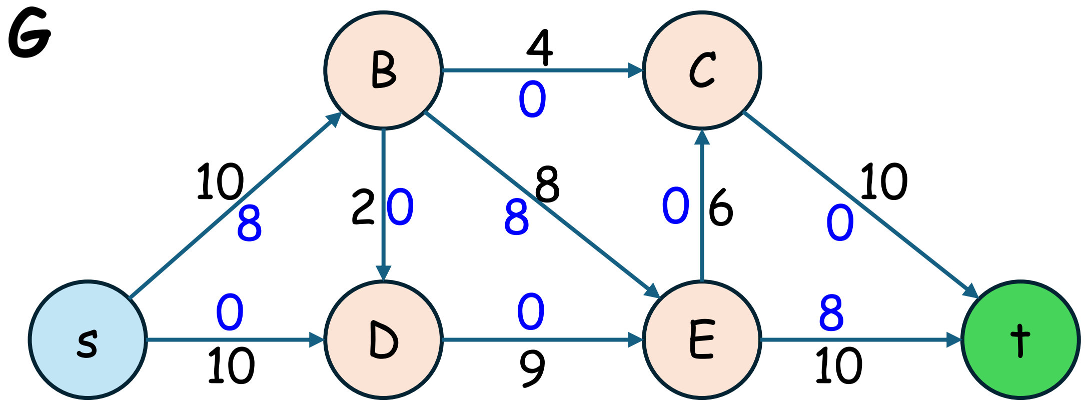


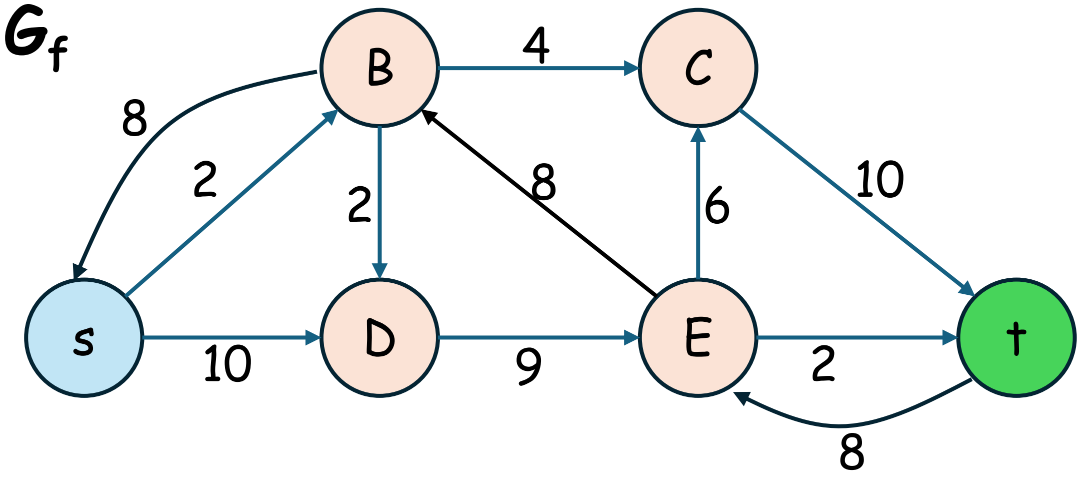


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
    f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
    f ← Augment(f, P)
    G_f ← Residual-Graf(G)     // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
    if e ∈ G.E then
      f(e) ← f(e) + b
    else
      f(e^R) ← f(e^R) - b

  return f
```

Flow value = 8

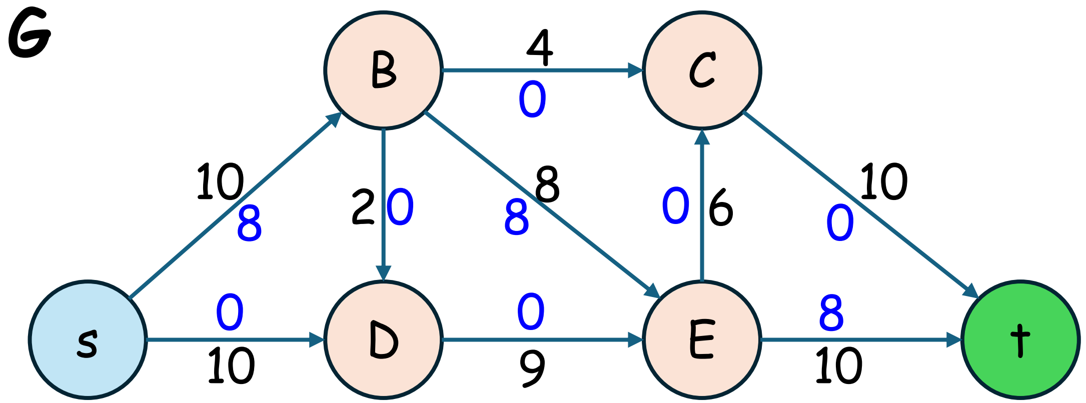


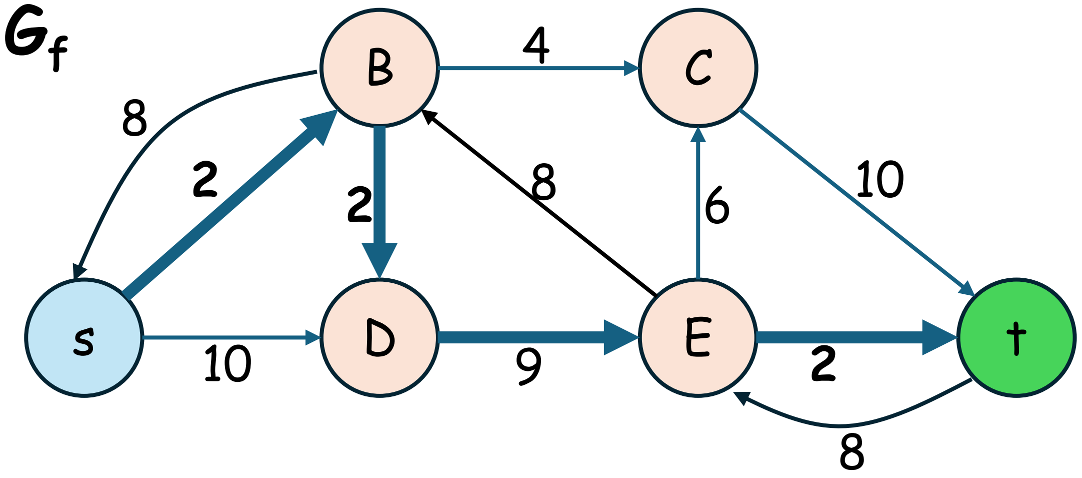


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
    f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
     f ← Augment(f, P)
     G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 8

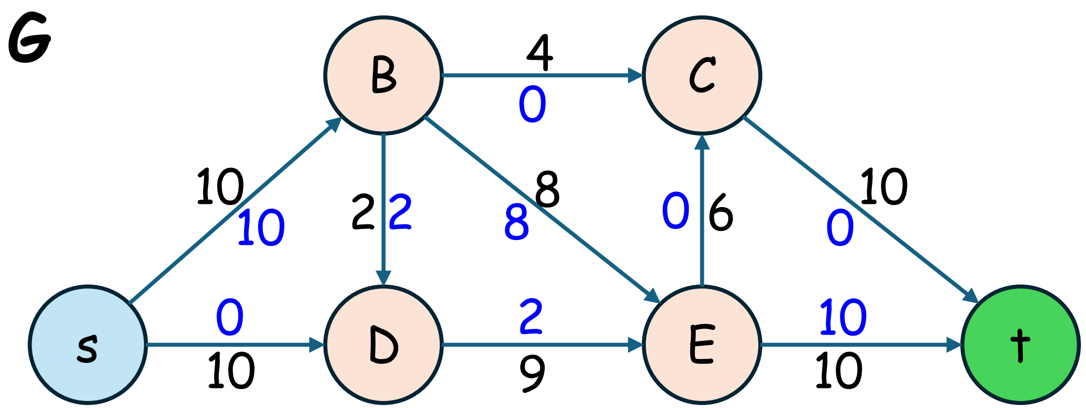


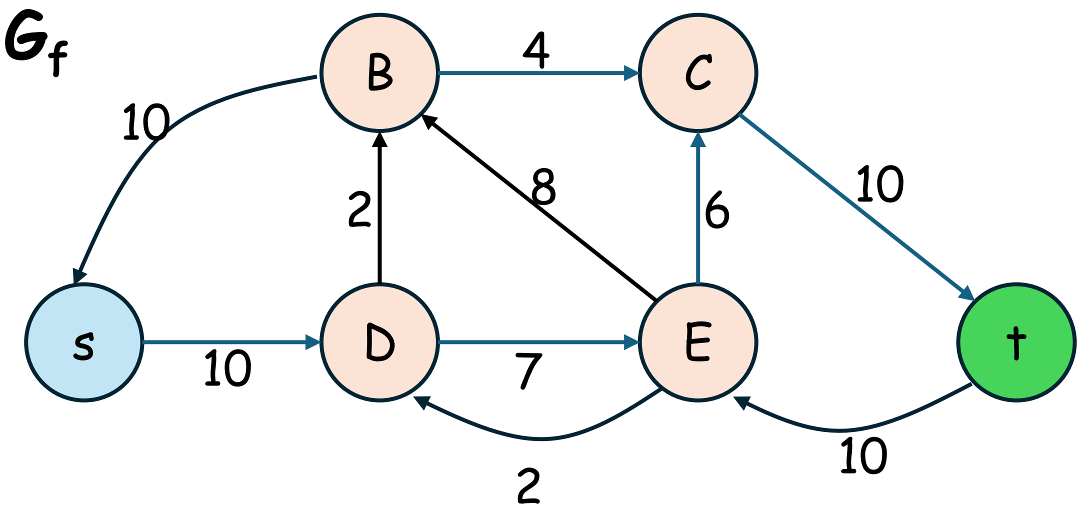


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
      f ← Augment(f, P)
      G_f ← Residual-Graf(G)     // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 10

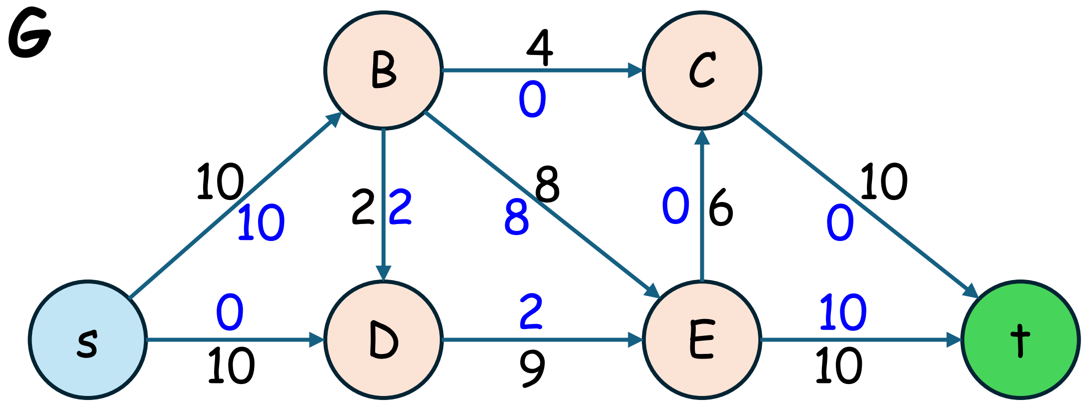


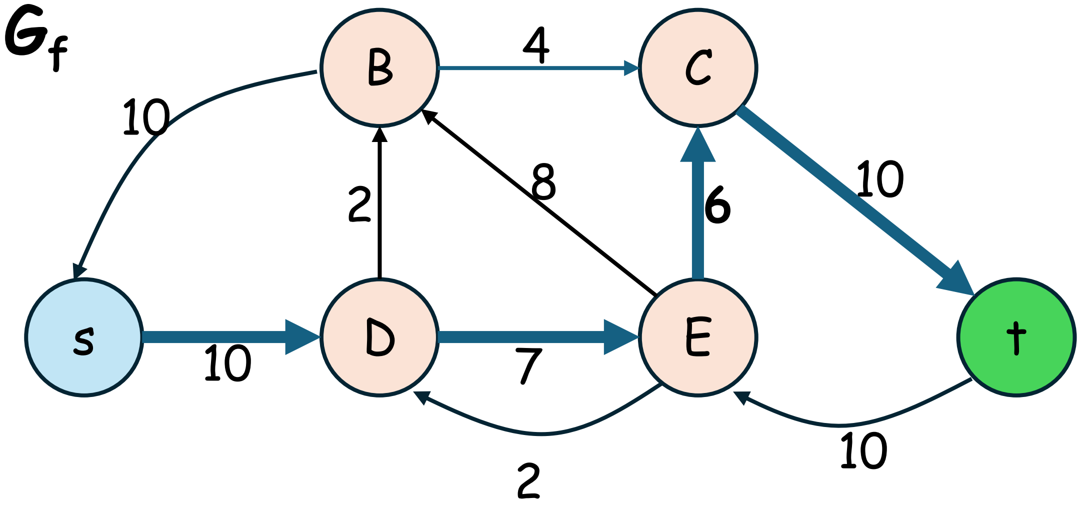


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
     f ← Augment(f, P)
     G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 10

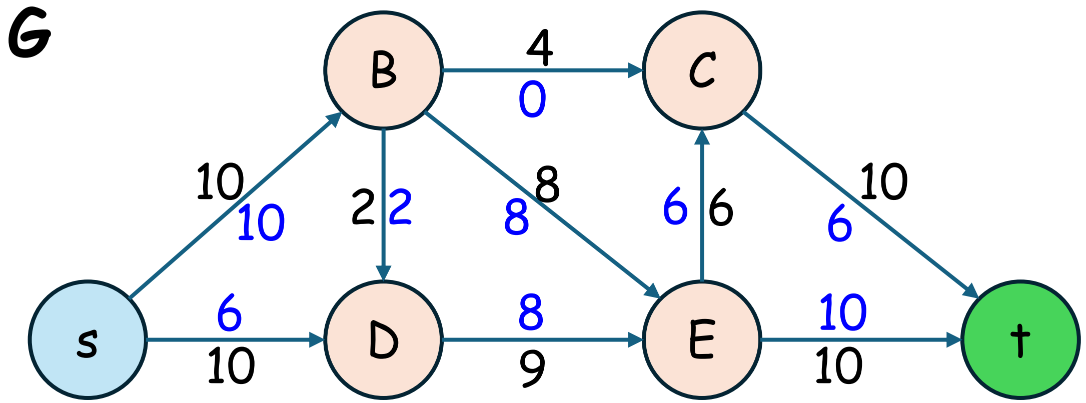


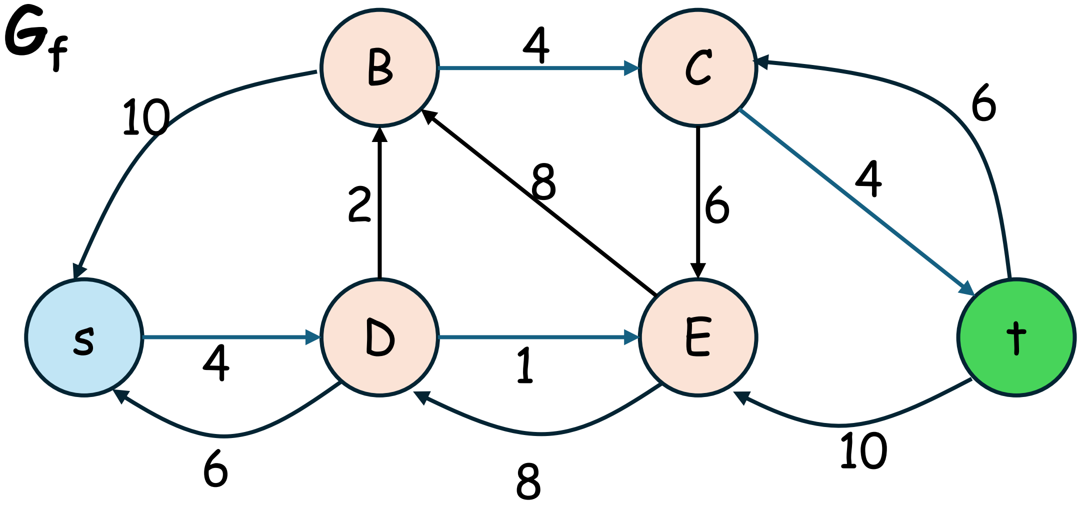


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
      f ← Augment(f, P)
      G_f ← Residual-Graf(G)     // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 16

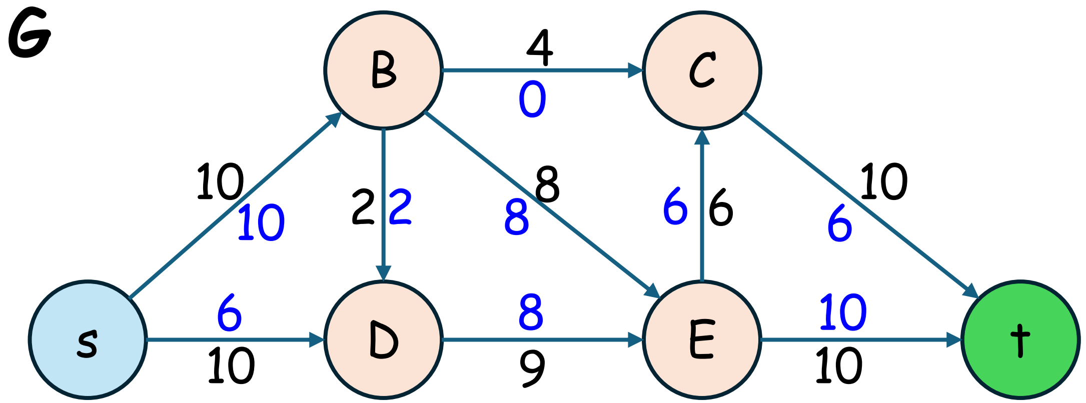


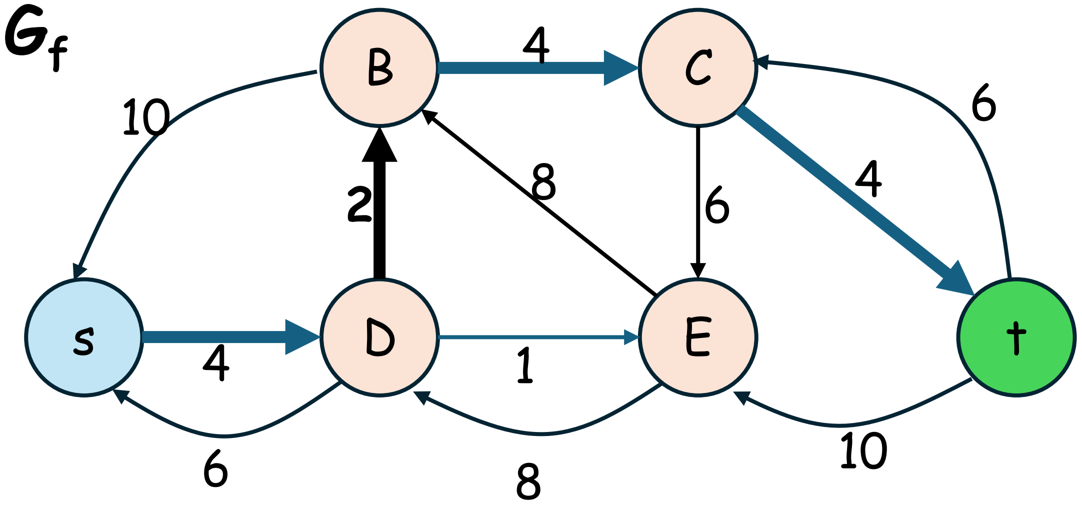


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
     f ← Augment(f, P)
     G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 16

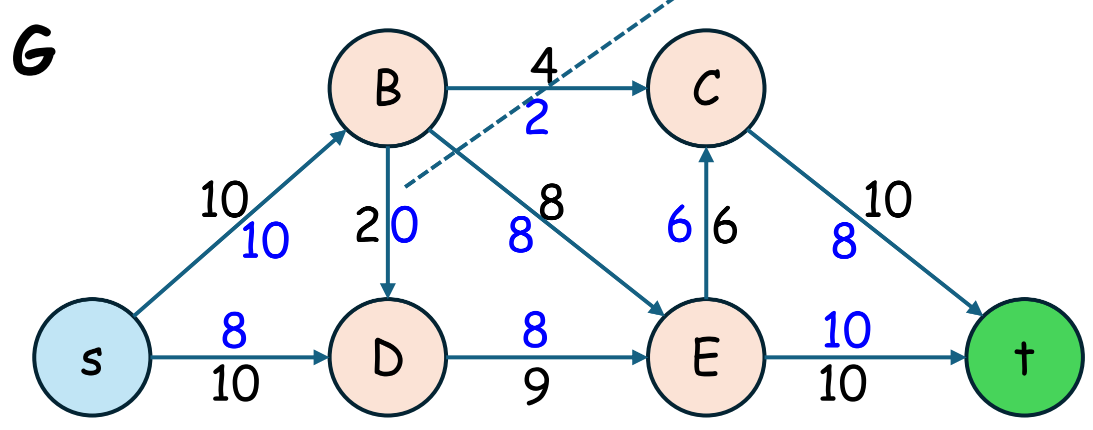


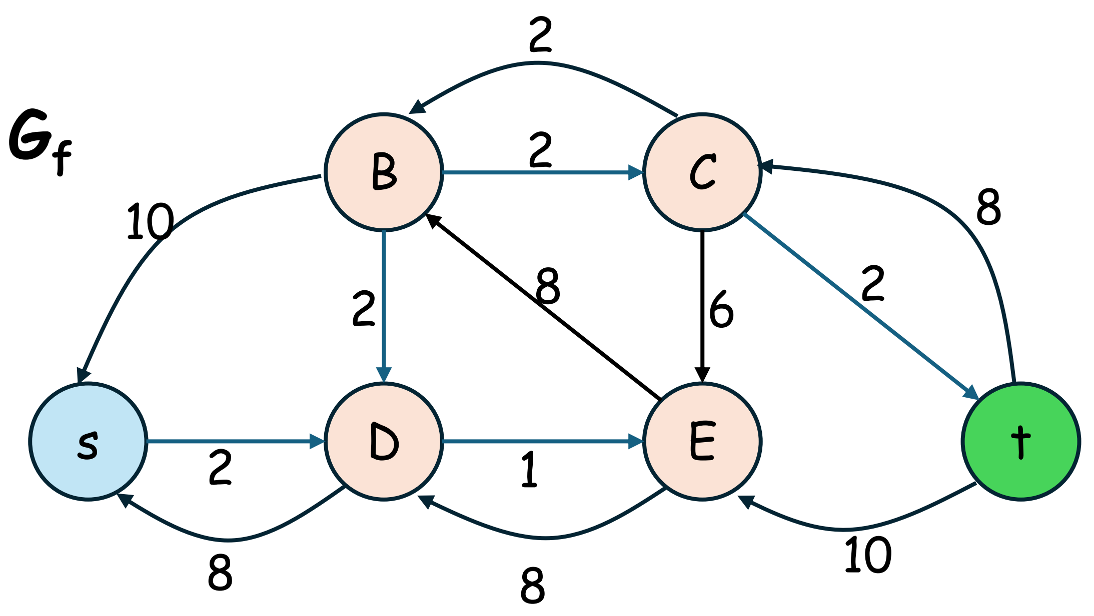


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  Gf ← Residual-Graf(G)
  while there exist augmenting s-t path P on Gf:
     f ← Augment(f, P)
     Gf ← Residual-Graf(G)     // update Gf

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 18

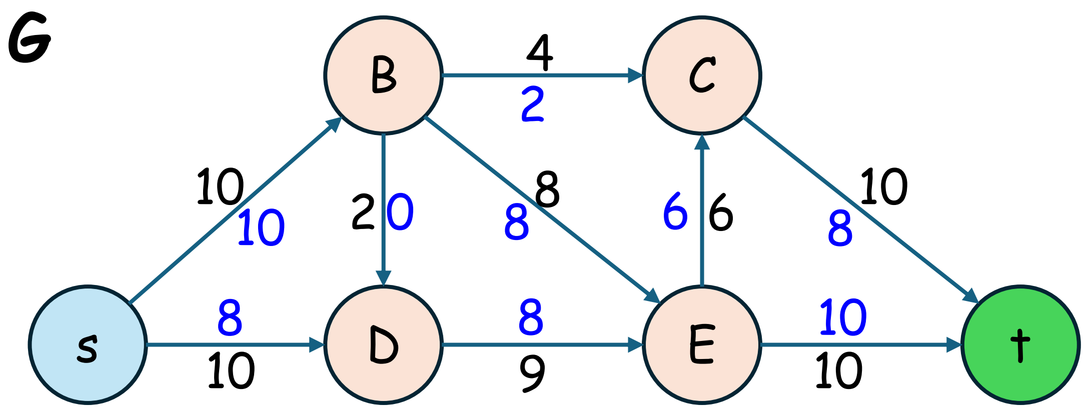


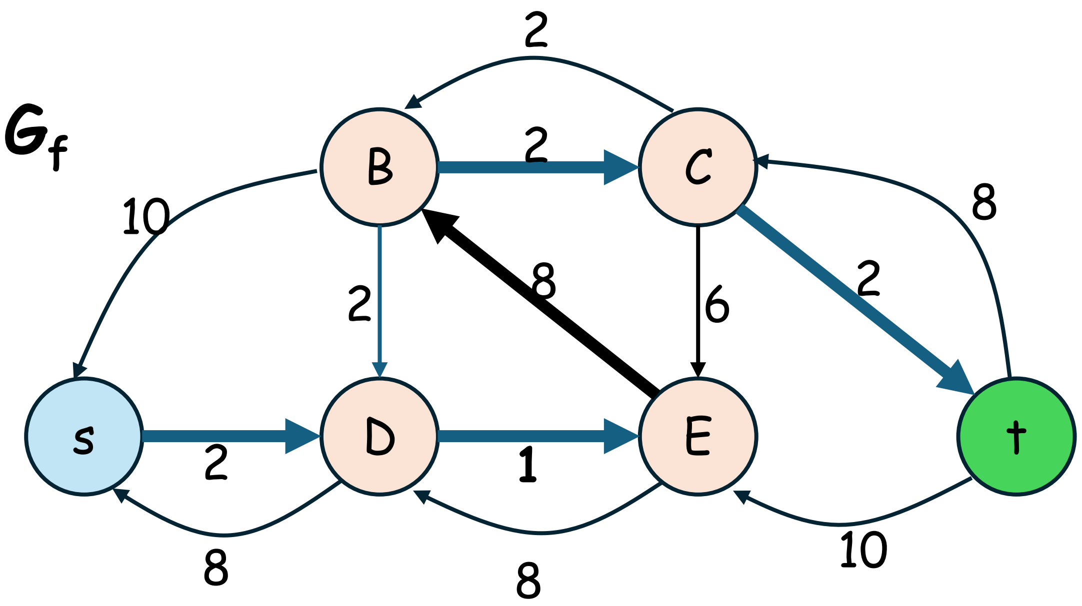


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
    f(e) ← 0

  G_f ← Residual-Graf(G)
  while there exist augmenting s-t path P on G_f:
    f ← Augment(f, P)
    G_f ← Residual-Graf(G)    // update G_f

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
    if e ∈ G.E then
      f(e) ← f(e) + b
    else
      f(e^R) ← f(e^R) - b

  return f
```

Flow value = 18

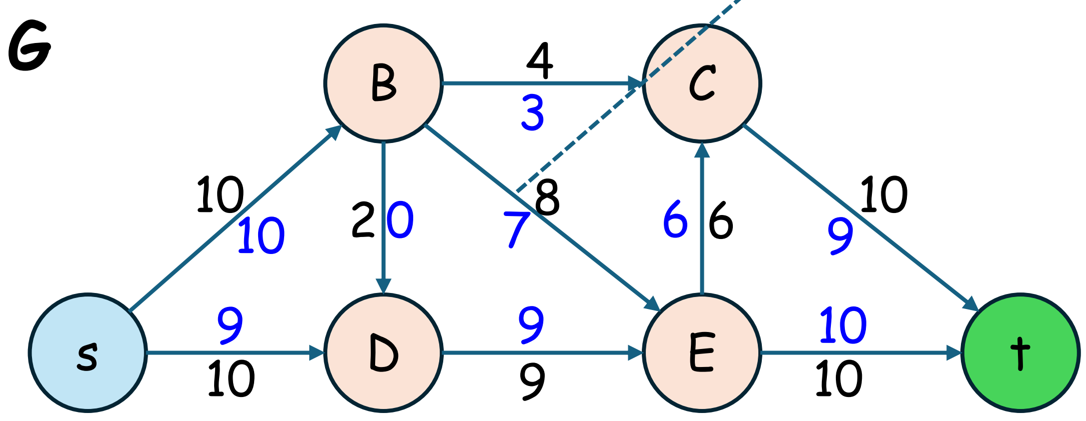


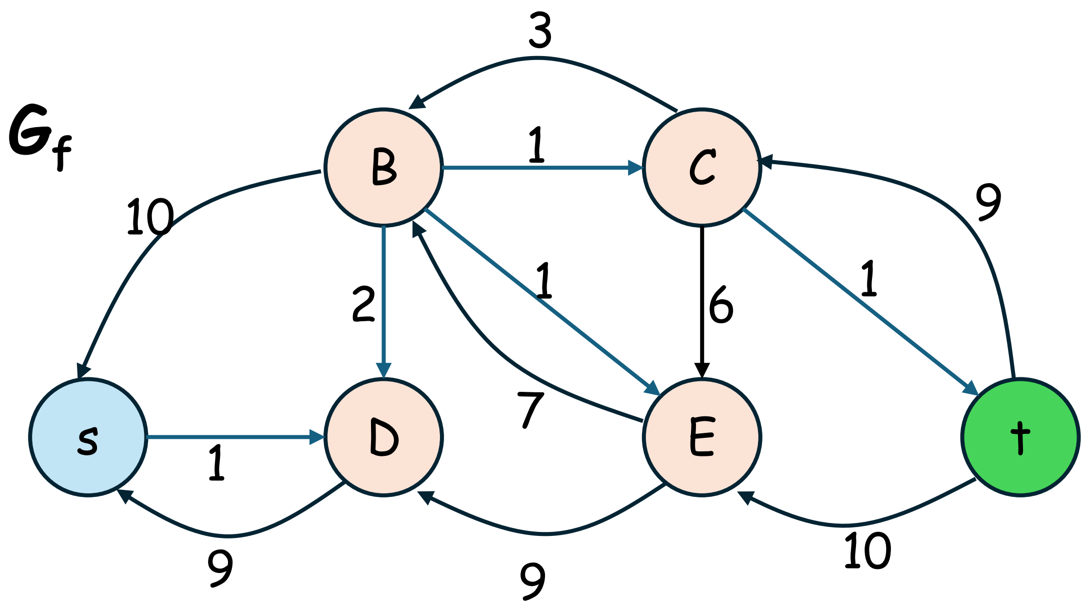


```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  Gf ← Residual-Graf(G)
  while there exist augmenting s-t path P on Gf:
      f ← Augment(f, P)
      Gf ← Residual-Graf(G)     // update Gf

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 19



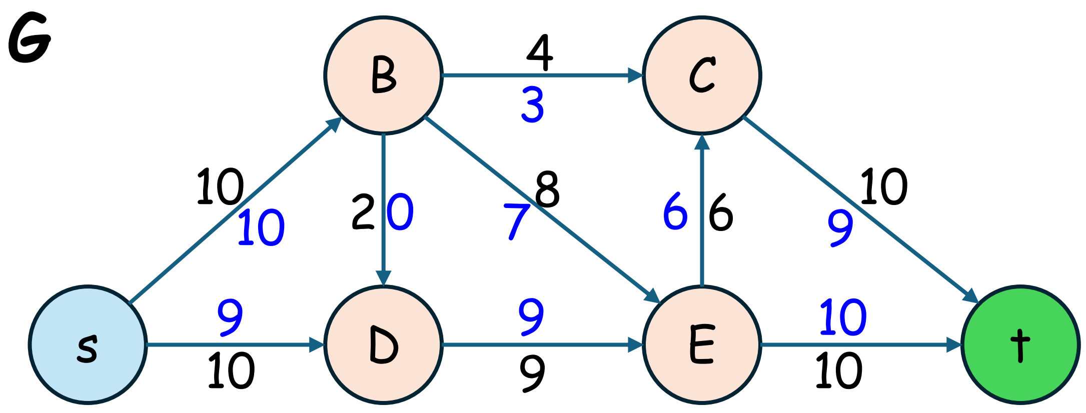


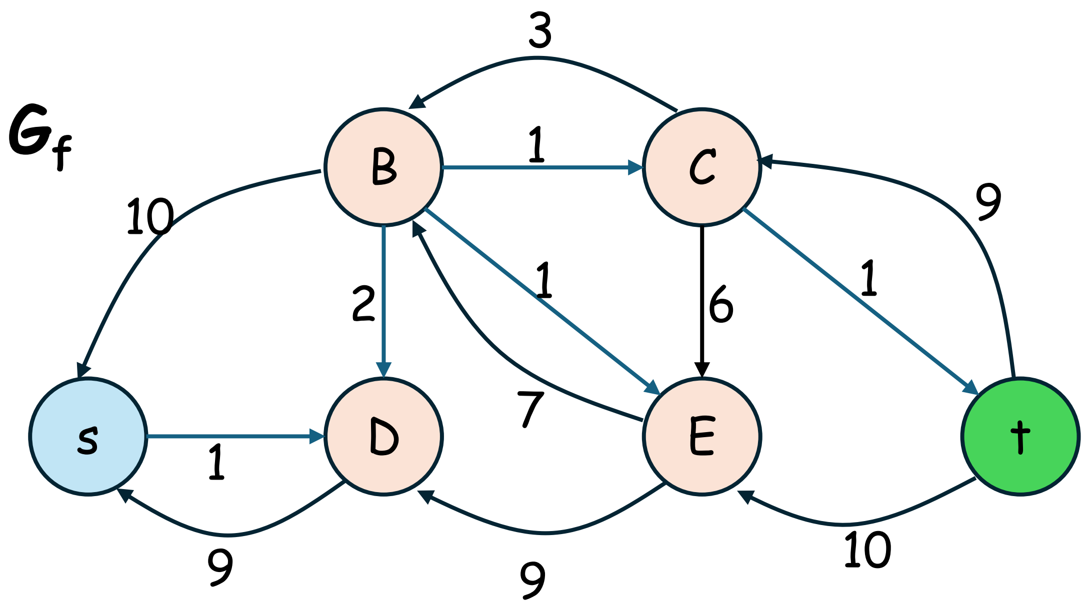
```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  Gf ← Residual-Graf(G)
  while there exist augmenting s-t path P on Gf:
     f ← Augment(f, P)
     Gf ← Residual-Graf(G)    // update Gf

  return f
```

False

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(e^R) ← f(e^R) - b

  return f
```

Flow value = 19



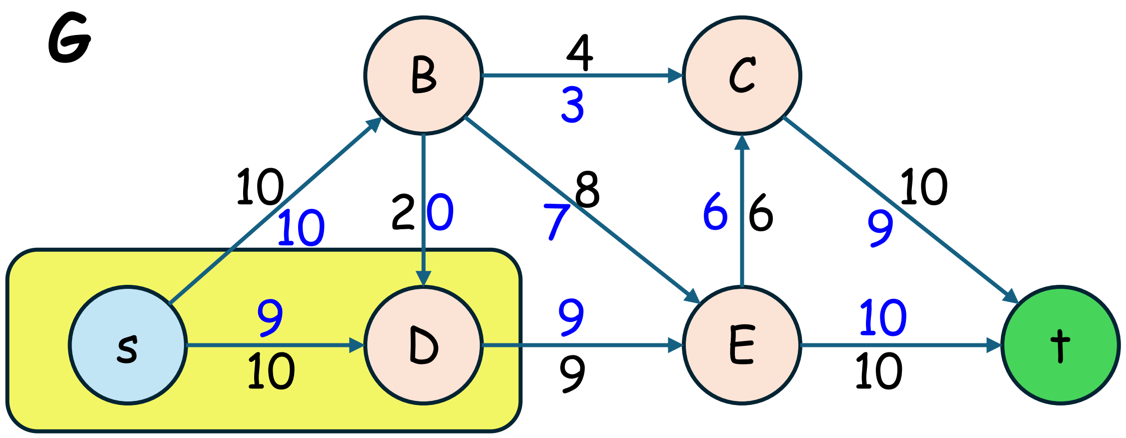


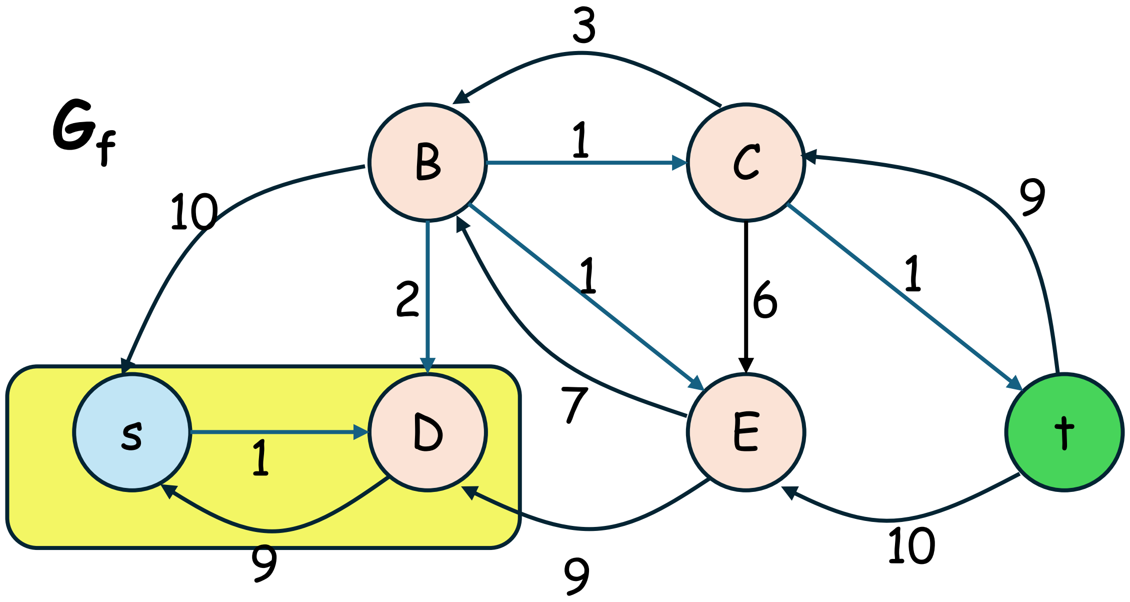
```
Fold-Fulkerson(G, s, t):
  for e ∈ G.E :
     f(e) ← 0

  Gf ← Residual-Graf(G)
  while there exist augmenting s-t path P on Gf:
     f ← Augment(f, P)
     Gf ← Residual-Graf(G)    // update Gf

  return f
```

```
Augment(f, P):
  b ← Bottleneck(P, f)

  for e ∈ P:
     if e ∈ G.E then
        f(e) ← f(e) + b
     else
        f(eR) ← f(eR) - b

  return f
```

Flow value = 19 (max)

Cut Capacity = 19





```c
int fordFulkerson(int graph[V][V], int s, int t) {
    int u, v;
    int residualGraph[V][V];

    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            residualGraph[u][v] = graph[u][v];

    int parent[V];
    int max_flow = 0;

    while (bfs(residualGraph, s, t, parent)) {
        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            path_flow = min(path_flow, residualGraph[u][v]);
        }

        for (v = t; v != s; v = parent[v]) {
            u = parent[v];
            residualGraph[u][v] -= path_flow;
            residualGraph[v][u] += path_flow;
        }
        max_flow += path_flow;
    }
    return max_flow;
}
```

Initially, residual Graph = Graph

This is useful for storing path; this is filled by **bfs()**

Bottleneck(P, f): Finding minimum residual capacity

Update residual graph

```
bool bfs(int residualGraph[V][V], int s, int t, int parent[]) {
    bool visited[V];
    memset(visited, 0, sizeof(visited));

    queue<int> q;
    q.push(s);
    visited[s] = true;
    parent[s] = -1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 0; v < V; v++) {
            if (visited[v] == false && residualGraph[u][v] > 0) {
                if (v == t) {
                    parent[v] = u;
                    return true;
                }
                q.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }
    return false;
}
```

BFS is used to find augmenting s-t path!

Correctness & Analysis

Yes, you are not taking “the DAA course” if you don’t learn these stuffs ☺

This is where the interesting part will start ☺

Max-Flow Min-Cut Theorem

Augmenting path theorem. Flow f is a max flow iff there are no augmenting paths.

Max-flow min-cut theorem. [Ford-Fulkerson 1956] The value of the max flow is equal to the value of the min cut.

Proof strategy. We prove both theorem simultaneously by showing the equivalence of the following conditions:

(i) There exists a cut (A^*, B^*) such that $v(f) = cap(A^*, B^*)$.
(ii) Flow f is a max flow.
(iii) There is no augmenting path relative to f.

(i) $\Rightarrow$ (ii) This was the corollary to weak duality lemma.

(ii) $\Rightarrow$ (iii) We show contrapositive.

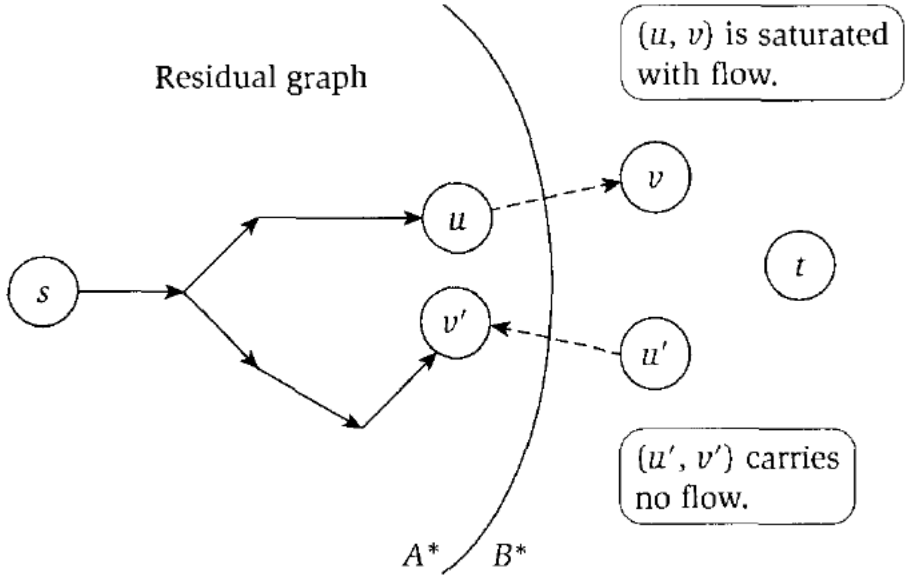
- Let f be a flow. If there exists an augmenting path, then we can improve f by sending flow along the path.

Proof of Max-Flow Min-Cut Theorem

(iii) $\Rightarrow$ (i)

- Let f be a flow with no augmenting paths.
- Let A* be set of vertices reachable from s in residual graph.
- By definition of A*, $s \in A^*$.
- By definition of f, $t \notin A^*$ and $t \in B^*$.
- So, (A*,B*) is indeed an s-t cut.

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\ &= \sum_{e \text{ out of } A^*} c(e) - 0 \\ &= cap(A^*, B^*) \quad \blacksquare \end{aligned}$$



Running Time

Assumption. Let C denote $\sum_{e \text{ out of } s} c(e)$.

All capacities are integers between 1 and C.

Invariant. Every flow value f(e) and every residual capacities $c_f(e)$ remains an integer throughout the algorithm.

Theorem. The algorithm terminates in at most C iterations.
Proof. Each augmentation increase value by at least 1. ▪

m = |V| = banyaknya nodes

Theorem. The Ford-Fulkerson algorithm runs in O(mC) time.

Integrality theorem. If all capacities are integers, then there exists a max flow f for which every flow value f(e) is an integer.
Proof. Since algorithm terminates, theorem follows from invariant. ▪

Computing a minimum s-t cut from a maxflow

Theorem
Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time.

Proof
We simply follow the construction in the proof of the Max-Flow Min-Cut Theorem. We construct the residual graph G_f, and perform breadth-first search or depth-first search to determine the set A^* of all nodes that s can reach. We then define $B^* = V - A^*$, and return the cut (A^*, B^*).

■

Ford-Fulkerson --> $O(mC)$

- Do you think that **$O(mC)$** is polynomial?

Ford-Fulkerson --> $O(mC)$

- Do you think that **$O(mC)$** is polynomial?

- No, it's **exponential** in terms of input size!
- C is integer, and the number of bits for representing C is **$\log_2 C$ = size(C)**.

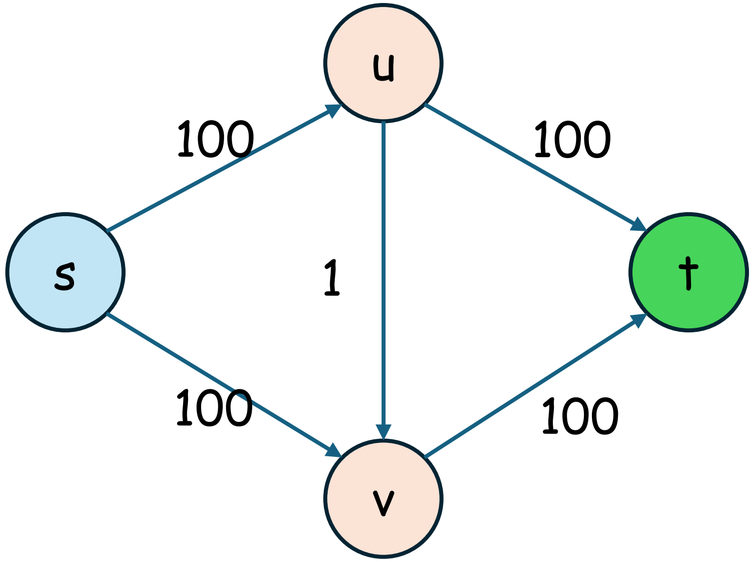
- $O(mC)$ = **$O(m2^{size(C)})$**

Bonus Stuffs

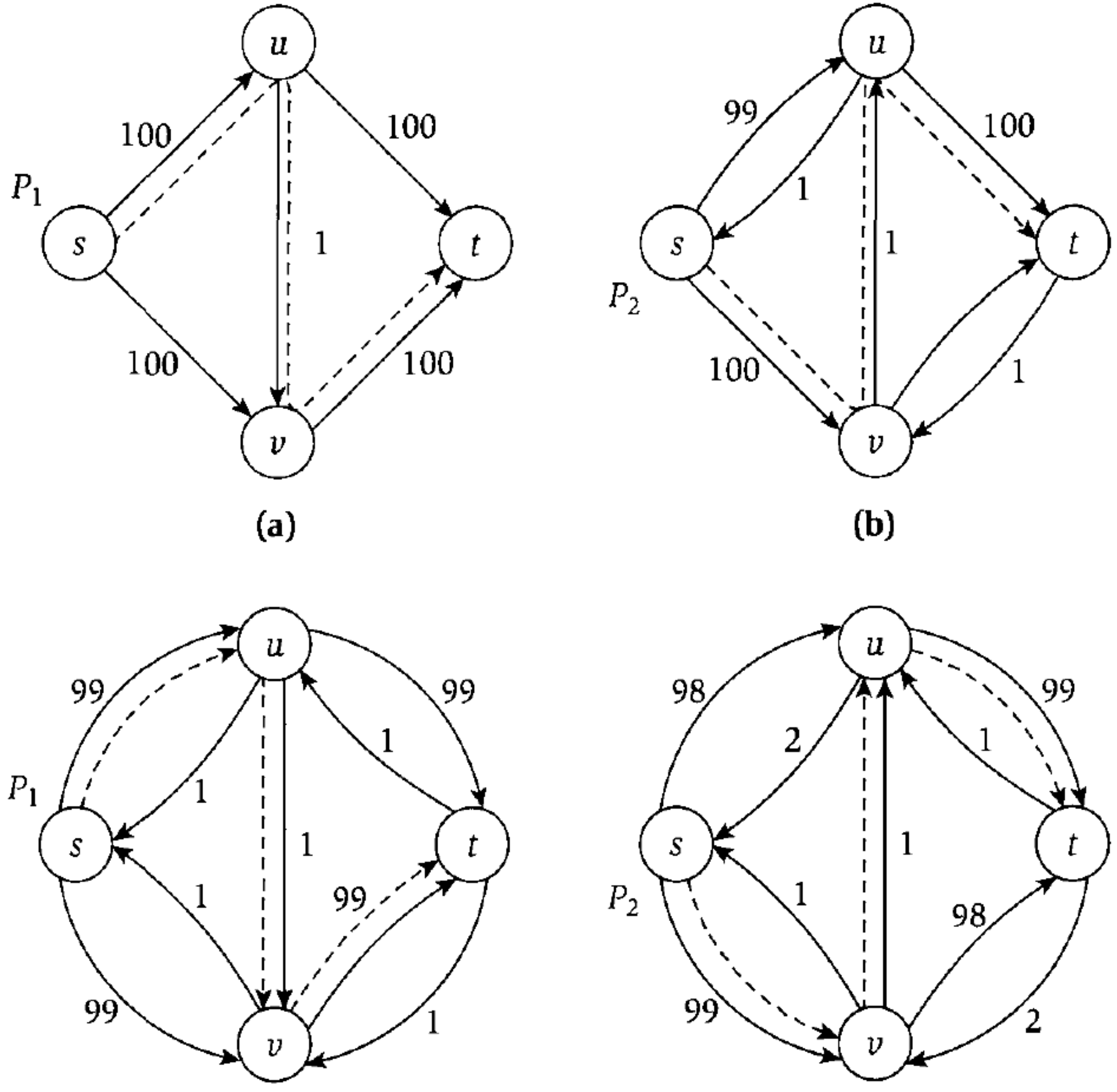
Can we still improve the algorithm?

Choosing Good Augmenting Paths

- In the Ford-Fulkerson Algo., any way of choosing an augmenting path increases the value of the flow;
- How to select augmenting paths so as to avoid the potential bad behavior of the algorithm?
- To get a sense for how bad the algorithm can be with pathological choices for the augmenting paths, consider the following graph:



(a) through (d) depict four iterations of the Ford-Fulkerson Algorithm using a **bad choice of augmenting paths**: The augmentations alternate between the path P_1 through nodes s, u, v, t and the path P_2 through the nodes s, v, u, t



(a) (b) (c) (d)

- It is easy to see that the maximum flow has value 200, and has f(e) = 100 for the edges (s, v), (s, u), (v,t) and (u,t) and value 0 on the edge (u, v). This flow can be obtained by a sequence of two augmentations, using the paths of nodes s, u, t and path s, v, t.
-
- Suppose we start with augmenting path P1 of nodes s, u, v, t . This path has bottleneck(P_1, f) = 1. After this augmentation, we have f(e) = 1 on the edge e = (u, v), so the reverse edge is in the residual graph.
- For the next augmenting path, we choose the path P_2 of the nodes s, v, u, t. In this second augmentation, we get bottleneck(P_2, f) = 1 as well. After this second augmentation, we have f(e) = 0 for the edge e = (u, v), so the edge is again in the residual graph.
- Suppose we alternate between choosing P_1 and P_2 for augmentation. In this case, each augmentation will have 1 as the bottleneck capacity, and it will take 200 augmentations to get the desired flow of value 200.

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.

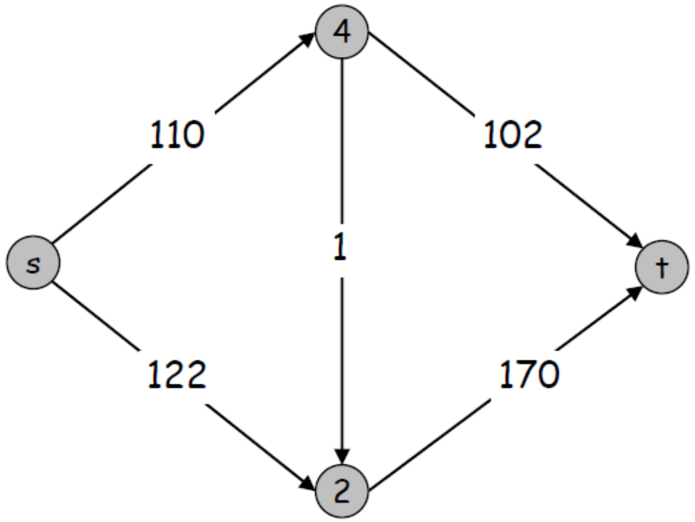
Choose augmenting paths with: [Edmonds-Karp 1972, Dinitz 1970]

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

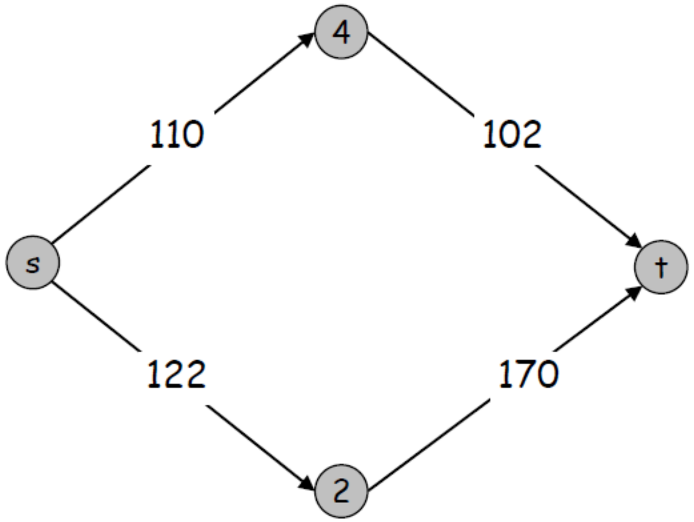
Capacity Scaling

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 edges with capacity at least Δ.



G_f



$G_f(100)$

Ford-Fulkerson Algorithm with Capacity Scaling

```
Scaling Max-Flow
  Initially f(e) = 0 for all e in G
  Initially set Δ to be the largest power of 2 that is no larger
        than the maximum capacity out of s: Δ ≤ max_{e out of s} c_e
    While Δ ≥ 1
      While there is an s-t path in the graph G_f(Δ)
        Let P be a simple s-t path in G_f(Δ)
        f' = augment(f, P)
        Update f to be f' and update G_f(Δ)
      Endwhile
      Δ = Δ/2
    Endwhile
Return f
```

Theorem:
The scaling max-flow algo. finds a max flow in **O(m log C)** augmentations. It can be implemented to run in **O(m² log C)** time.