# Network Flows

# Network Flow

Imagine that you are a courier service, and you want to deliver some cargo from one city to another.
You can deliver them using various flights from cities to cities, but each flight has a limited amount of space that you can use.

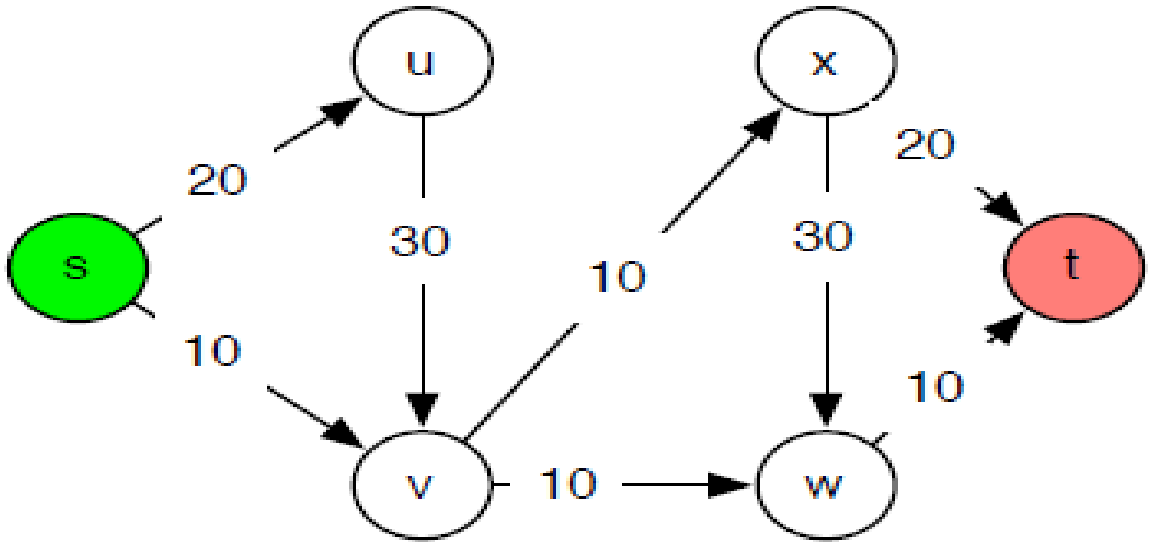
An important question is, how much of our cargo can be shipped to the destination using the different flights available?

To answer this question, we explore what is called a network flow graph, and show how we can model different problems using such a graph.

# Flow Network

A **flow network** is a connected, directed graph $G = (V, E)$.

- Each edge $e$ has a non-negative, integer **capacity** $c_e$.
- A single **source** $s \in V$.
- A single **sink** $t \in V$.
- No edge enters the source and no edge leaves the sink.



# Assumptions

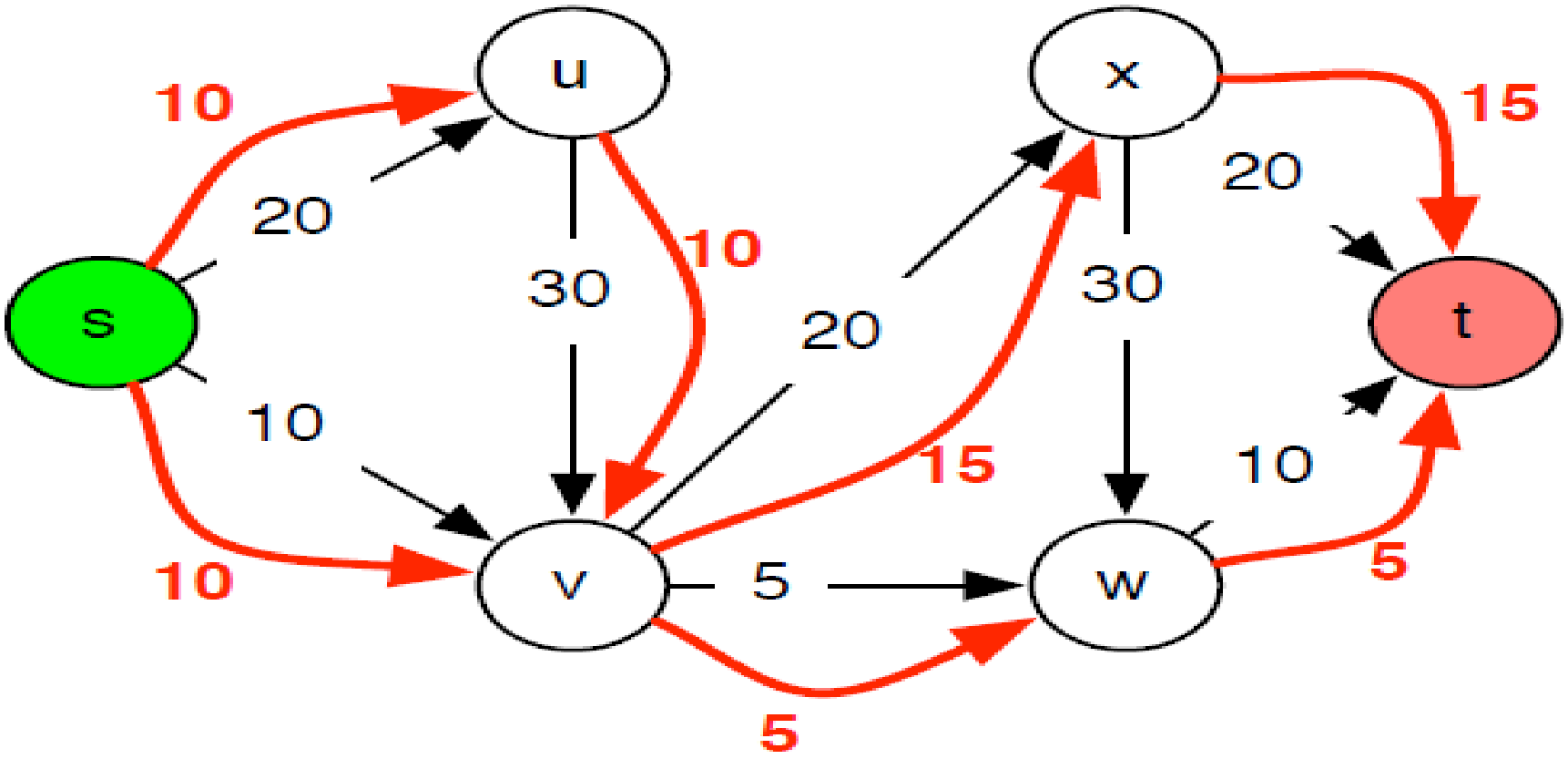
we make these assumptions about the network:

1. Capacities are integers.
2. Every node has one edge adjacent to it.
3. No edge enters the source and no edge leaves the sink.

# Flow

**Def.** An **s-t flow** is a function $f : E \longrightarrow \mathbb{R}^{\geq 0}$ that assigns a real number to each edge.

Intuitively, $f(e)$ is the amount of material carried on the edge $e$.



# Flow Constraints

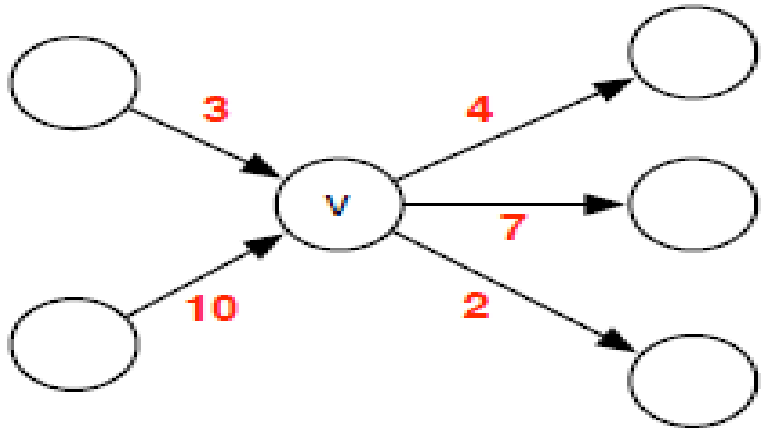
**Constraints on $f$:**

1. $0 \leq f(e) \leq c_e$ for each edge $e$. (capacity constraints)

2. For each node $v$ except $s$ and $t$, we have:

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

(balance constraints: whatever flows in, must flow out).



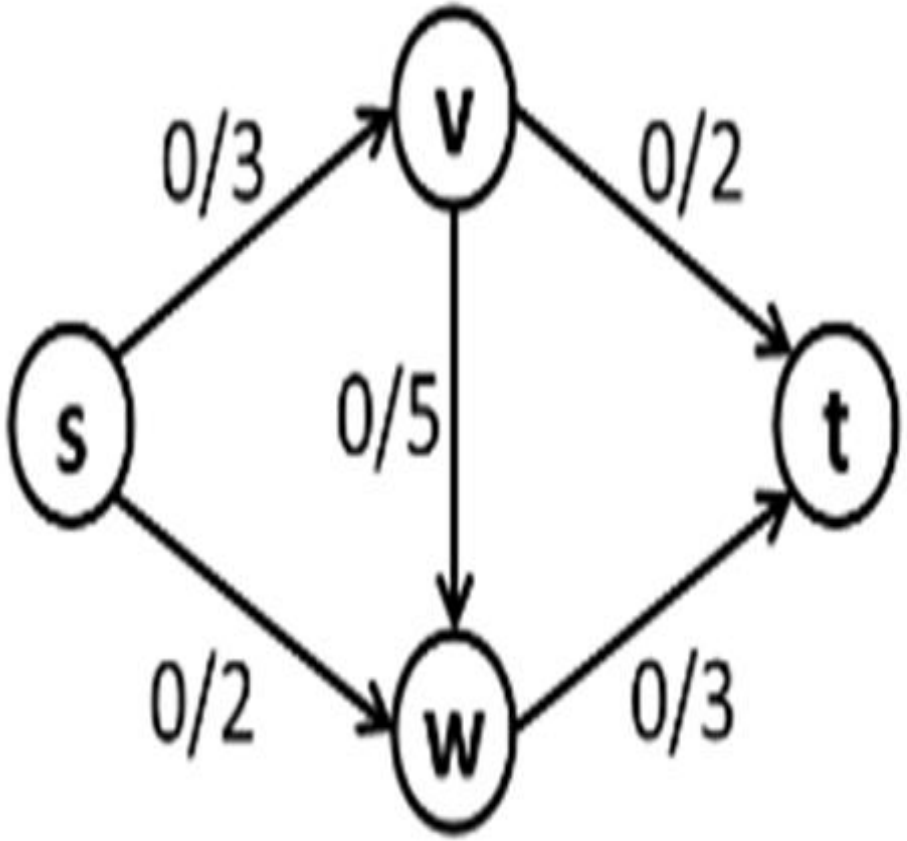
# Maximum flow problem

To achieve maximum transport between s and t is referred as maximum flow problem.

maximum flow problem: what is the largest flow of materials from source to sink that does not violate any capacity constraints?

# A Greedy Strategy

Suppose that we are trying to solve the maximum flow problem for the following network $G$ (where each label $f_e/c_e$ denotes both the flow $f_e$ pushed through an edge $e$ and the capacity $c_e$ of this edge):

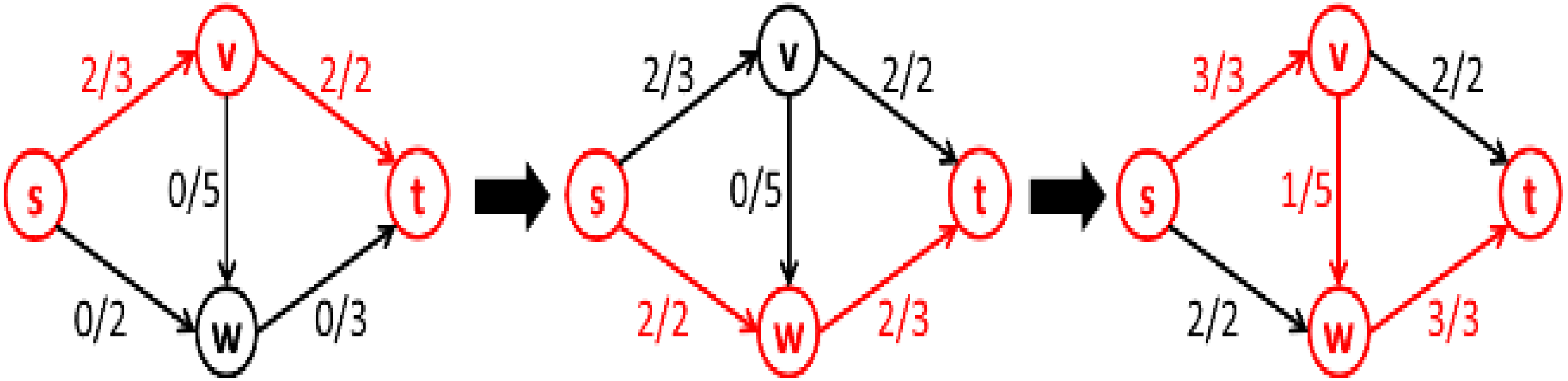


One possible greedy approach is the following:

1. Pick an arbitrary augmenting path $P$ that goes from the source vertex $s$ to the sink vertex $t$ such that $\forall e \left( e \in P \rightarrow f_e < c_e \right)$; that is, all of the edges in $P$ have available capacity.
2. Push the maximum possible flow $\Delta$ through this path. The value of $\Delta$ is determined by the *bottleneck* of $P$; that is, the edge with minimum available capacity. Formally, $\Delta = \min_{e \in P} \left( c_e - f_e \right)$.
3. Go to step 1 until no augmenting paths exist.

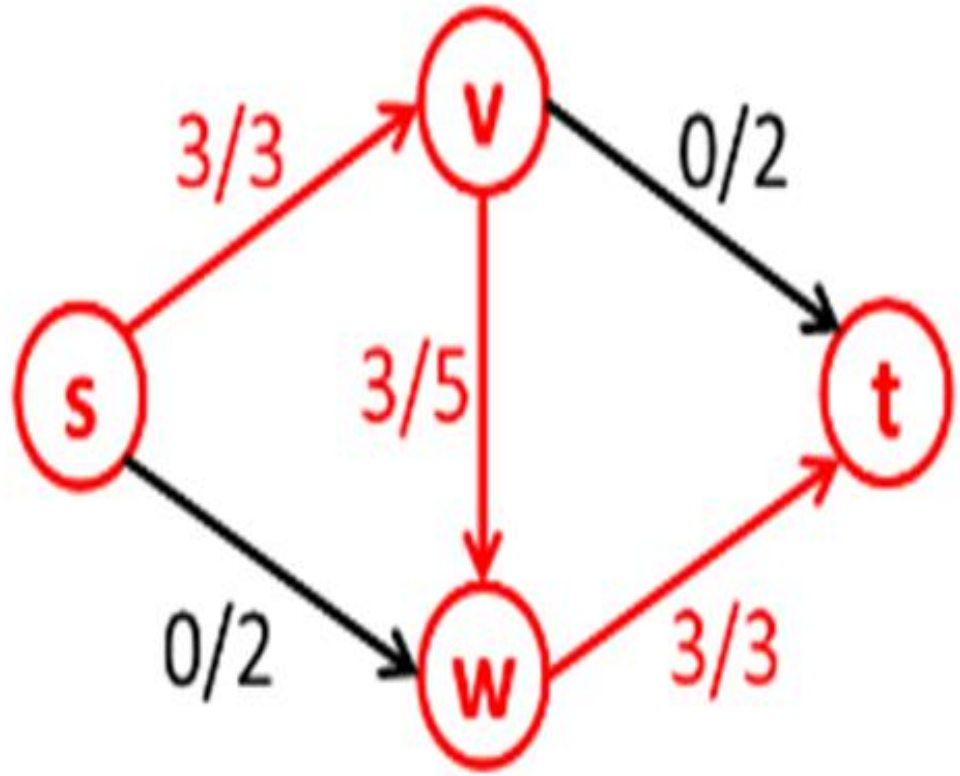
That is, find a path with available capacity, send flow along that path, and repeat.

In $G$, one possible execution of the above heuristic finds three augmenting paths $P_1$, $P_2$, and $P_3$, in this order. These paths push 2, 2, and 1 units of flow, respectively, for a total flow of 5:



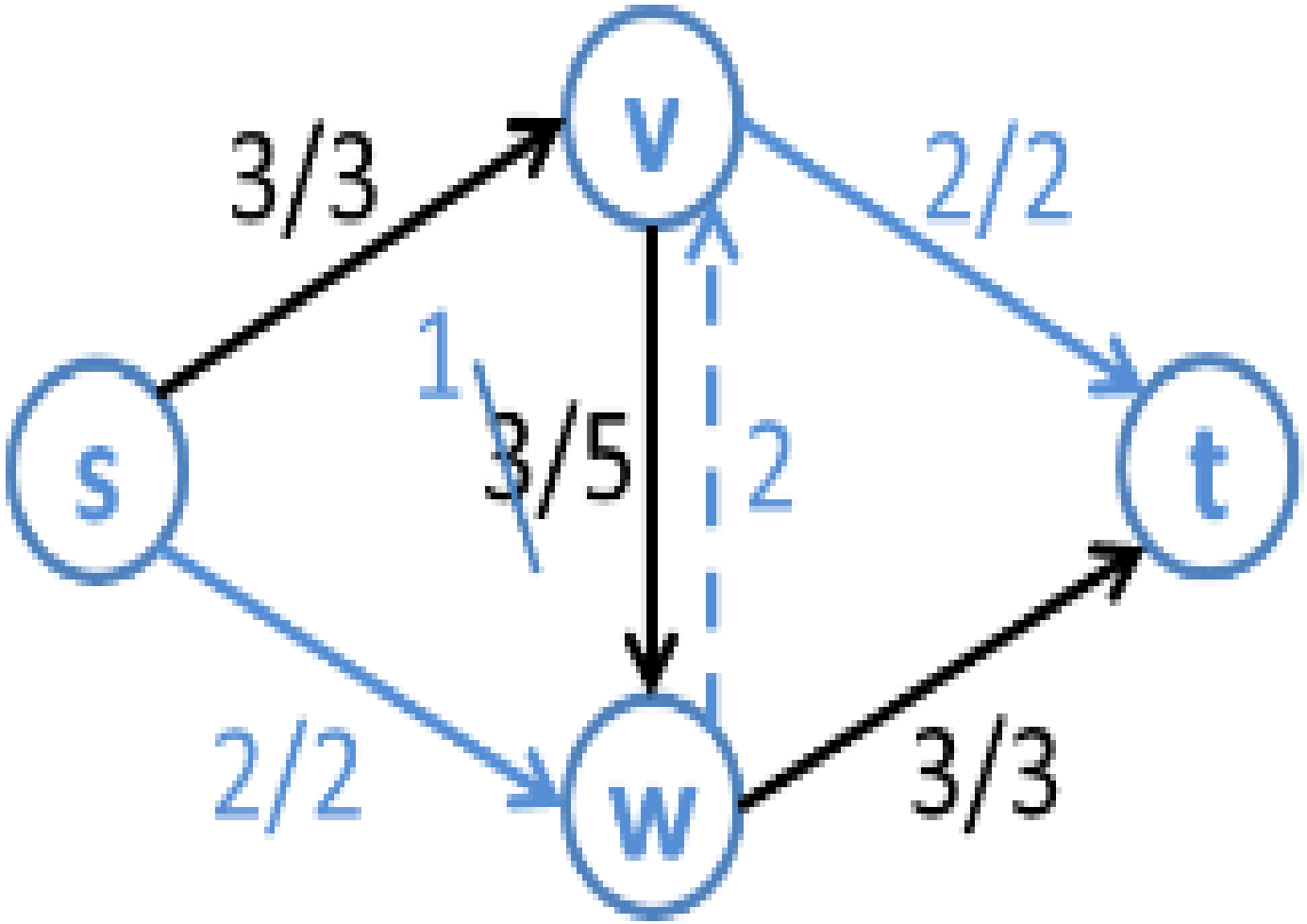

$P_1 = [s, v, t]$ $P_2 = [s, w, t]$ $P_3 = [s, v, w, t]$

Choosing paths in this order leads to an optimal solution; however, what happens if we select $P_3$ first (i.e., before $P_1$ and $P_2$)?




We get what is called a *blocking flow*: no more augmenting paths exist. In this case, the total flow is 3, which is not optimal. This issue can be resolved by allowing *undo* operations (i.e., by allowing flow to be sent in reverse, undoing work of previous iterations): simply push 2 units of flow backwards from vertex $w$ to vertex $v$ like this:

# Residual Graph



# Residual Graph

Encoding these allowed undo operations is the main goal of the **residual graph**.

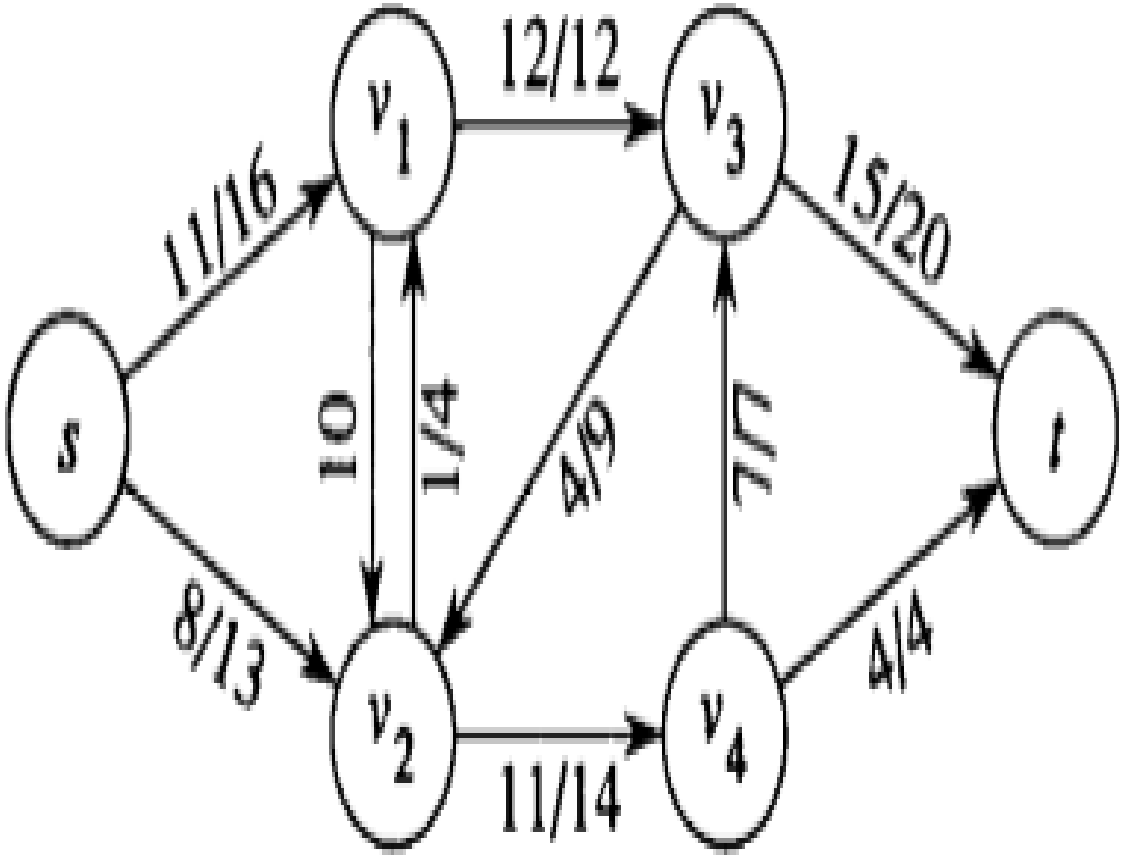
A residual graph $R$ of a network $G$ has the same set of vertices as $G$ and includes, for each edge $e = (u, v) \in G$ :

- A forward edge $e' = (u, v)$ with capacity $c_e - f_e$, if $c_e - f_e > 0$ .
- A backward edge $e'' = (v, u)$ with capacity $f_e$, if $f_e > 0$ .

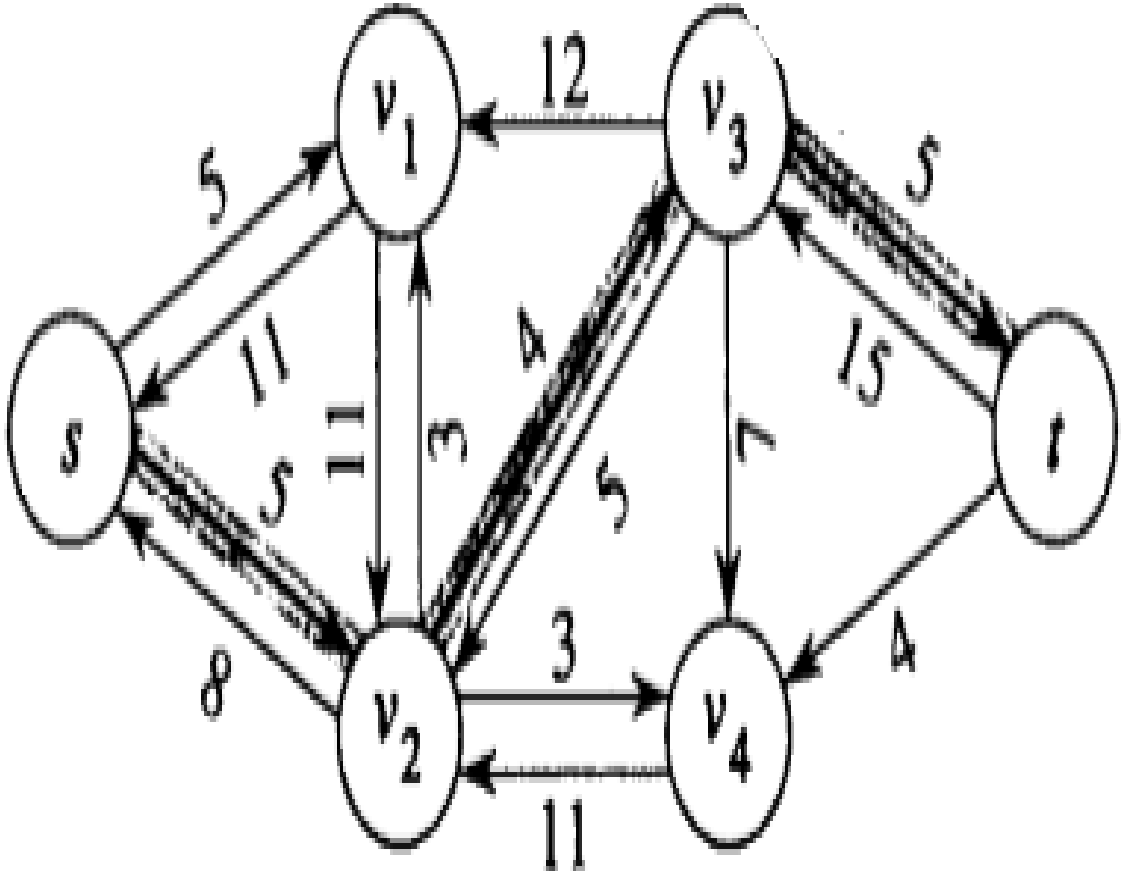
When an augmenting path $P'$ is selected in the residual graph $R$:

- Every edge in $P'$ that corresponds to a forward edge in $G$ increases the flow by using an edge with available capacity.
- Every edge in $P'$ that corresponds to an edge going backwards in $G$ undoes flow that was pushed in the forward direction in the past.

This is the main idea behind the Ford–Fulkerson method.




a flow




its residual network

# ford-fulkerson iterative method

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]
2      do f[u, v] ← 0
3         f[v, u] ← 0
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min {c_f(u, v) : (u, v) is in p}
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← −f[u, v]
```

Start by setting all flows to 0

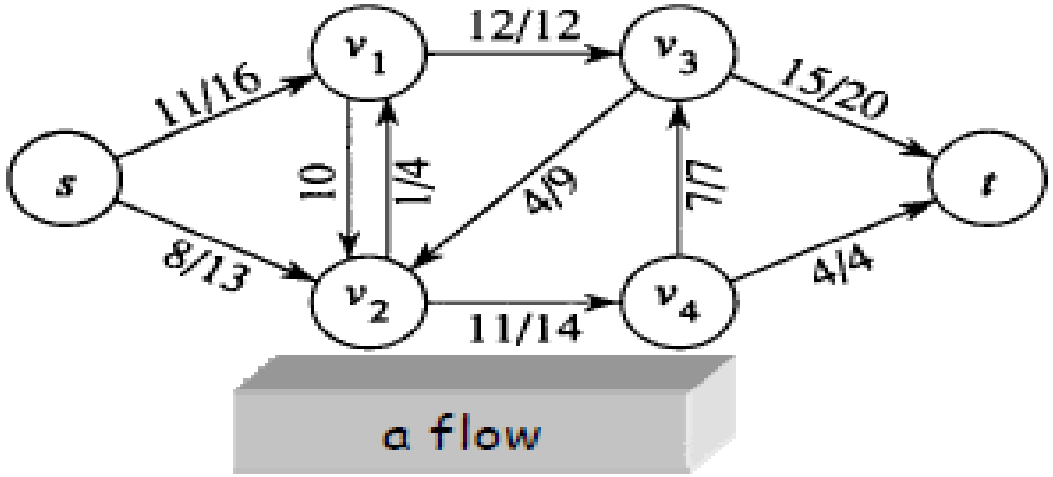
Pick some augmenting path $p$ which is a path in the residual network

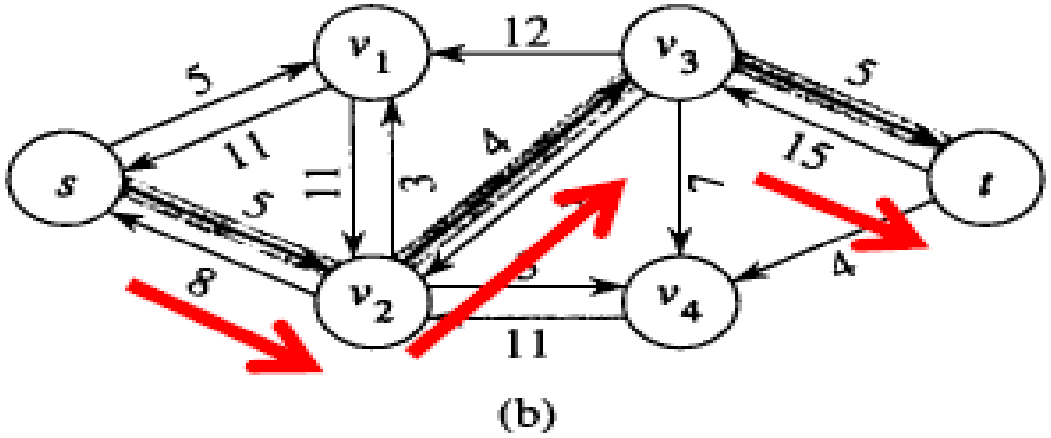
Determine its residual capacity

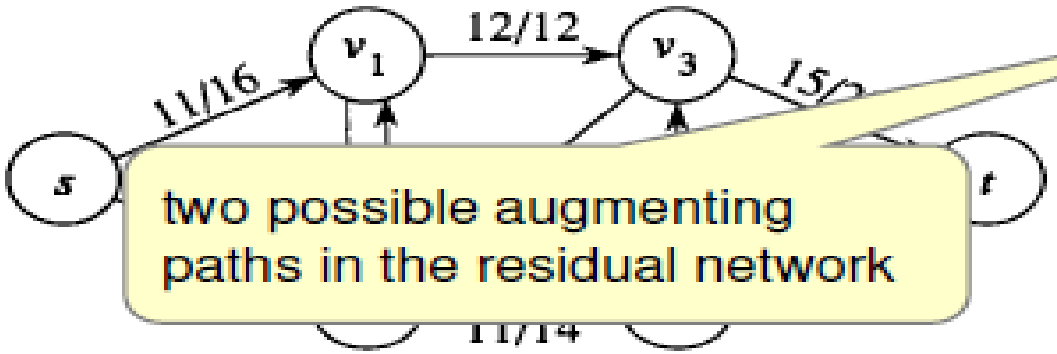
And increase the flow over the augmenting path by the residual capacity

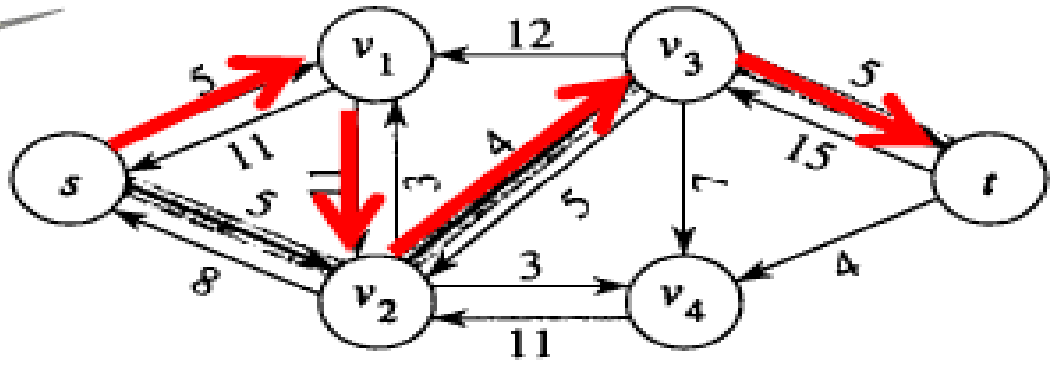
- the iteration stops when no augmenting path can be found.
- we will see that this results in a maximum flow.

- any arbitrary simple path from source to sink that we can find in the residual network shows a way to increase the flow in our network!
- this augmenting path is a simple path from $s$ to $t$ in the residual network.

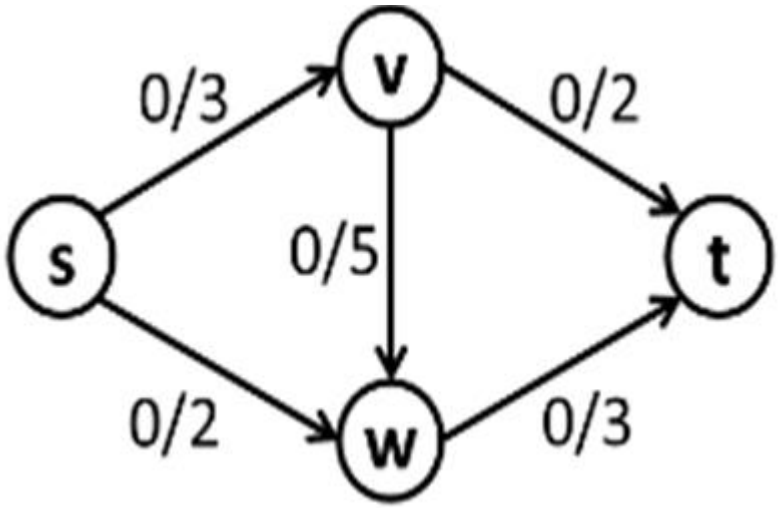


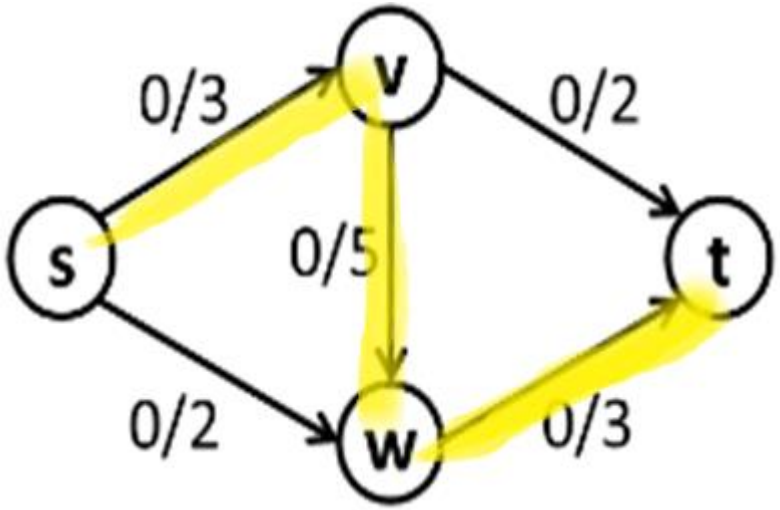



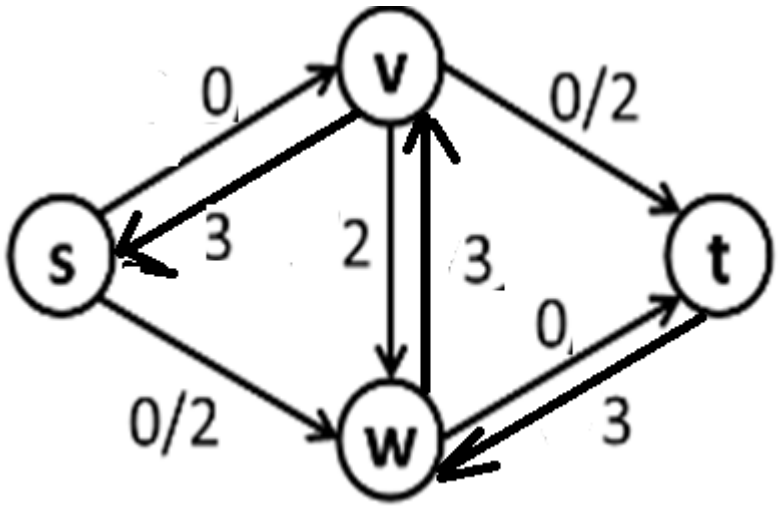



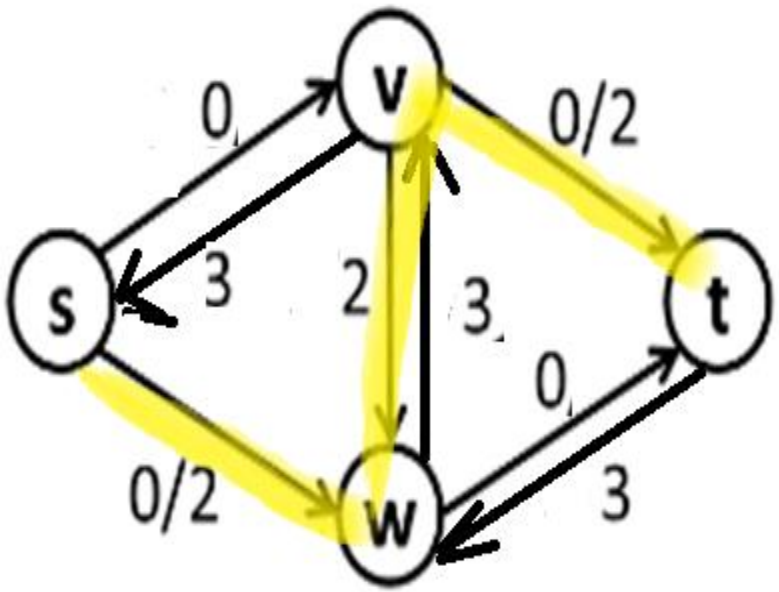
# Example: Apply Ford Fulkerson algorithm to find maximum flow.

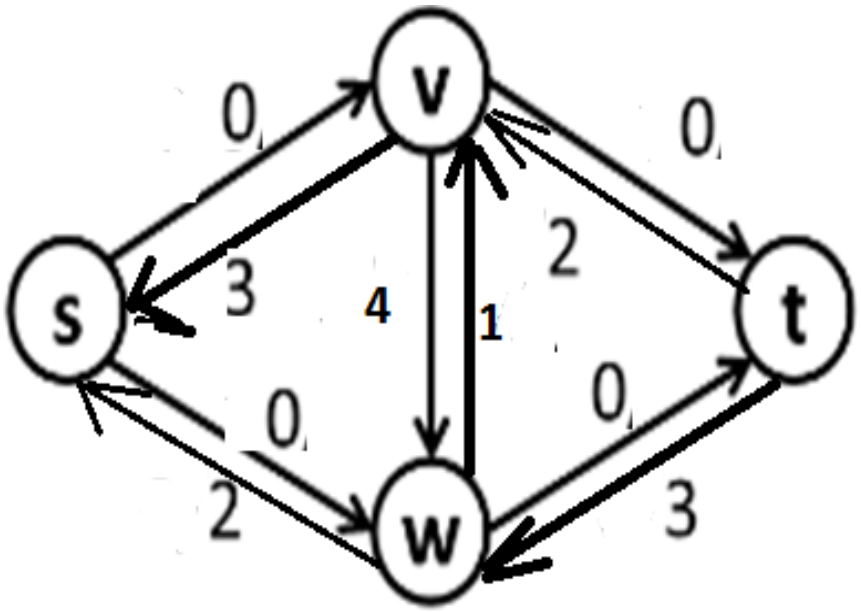





# Augmented path s-v-w-t

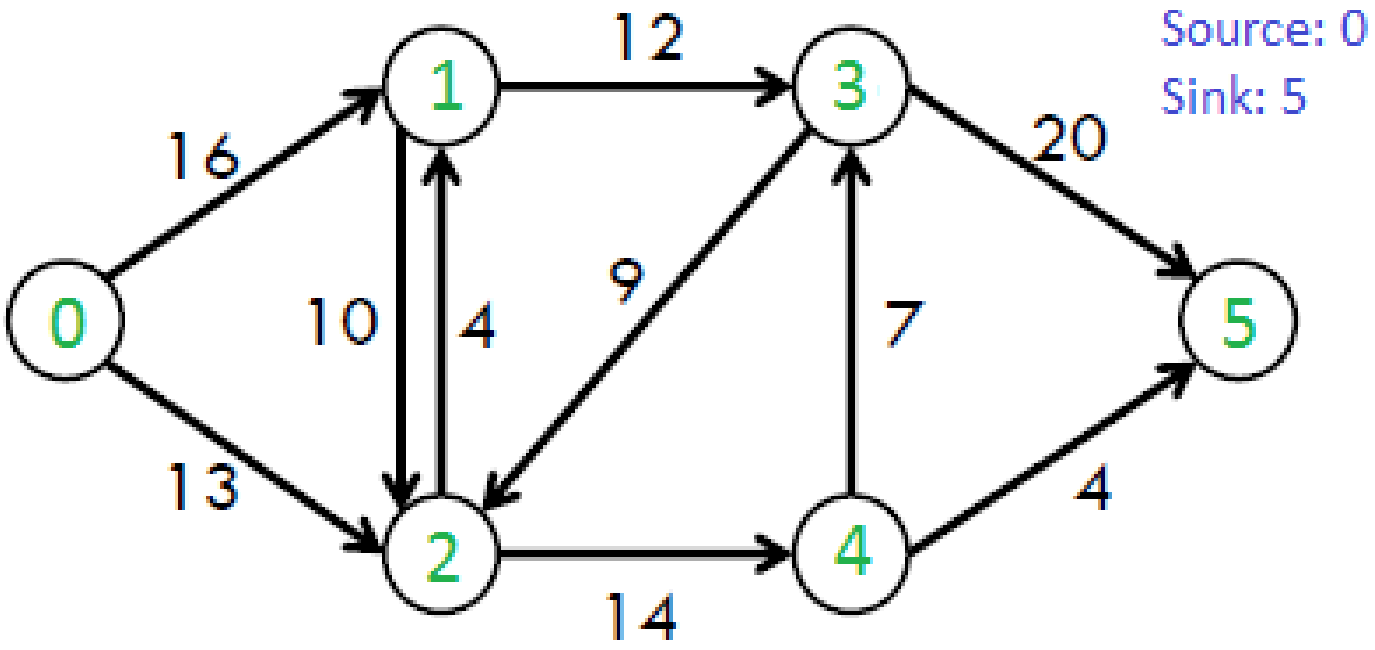


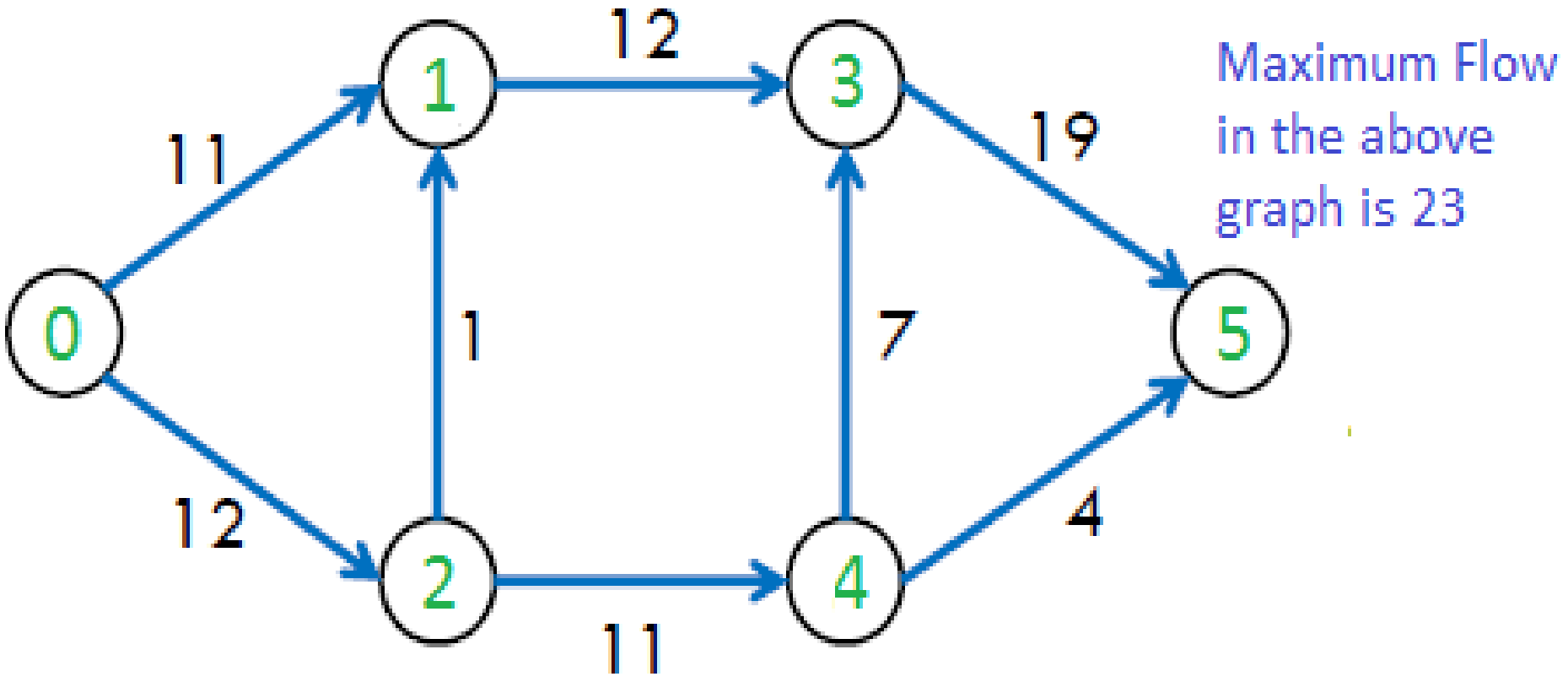

# Augmented path s-w-v-t

**So, Maximum Flow is 5**

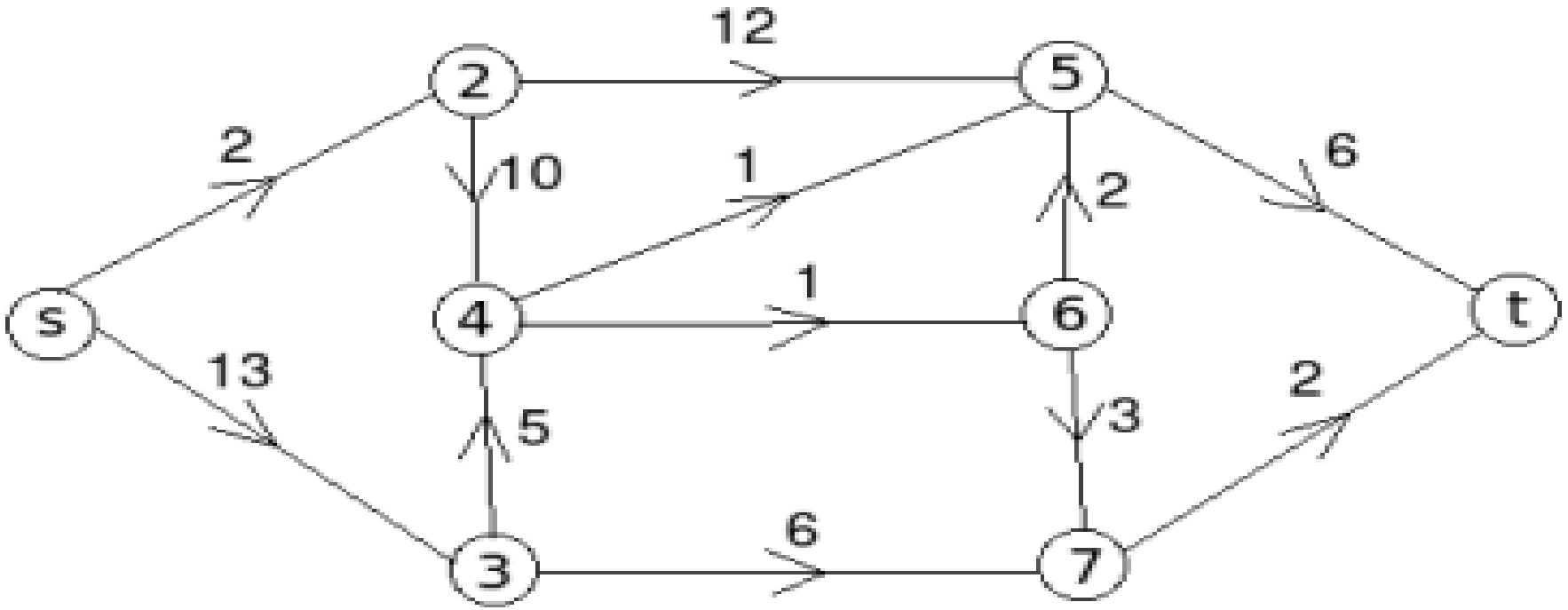
**Exercise1:** Apply Ford-Fulkerson algorithm to find maximum flow in the following network.



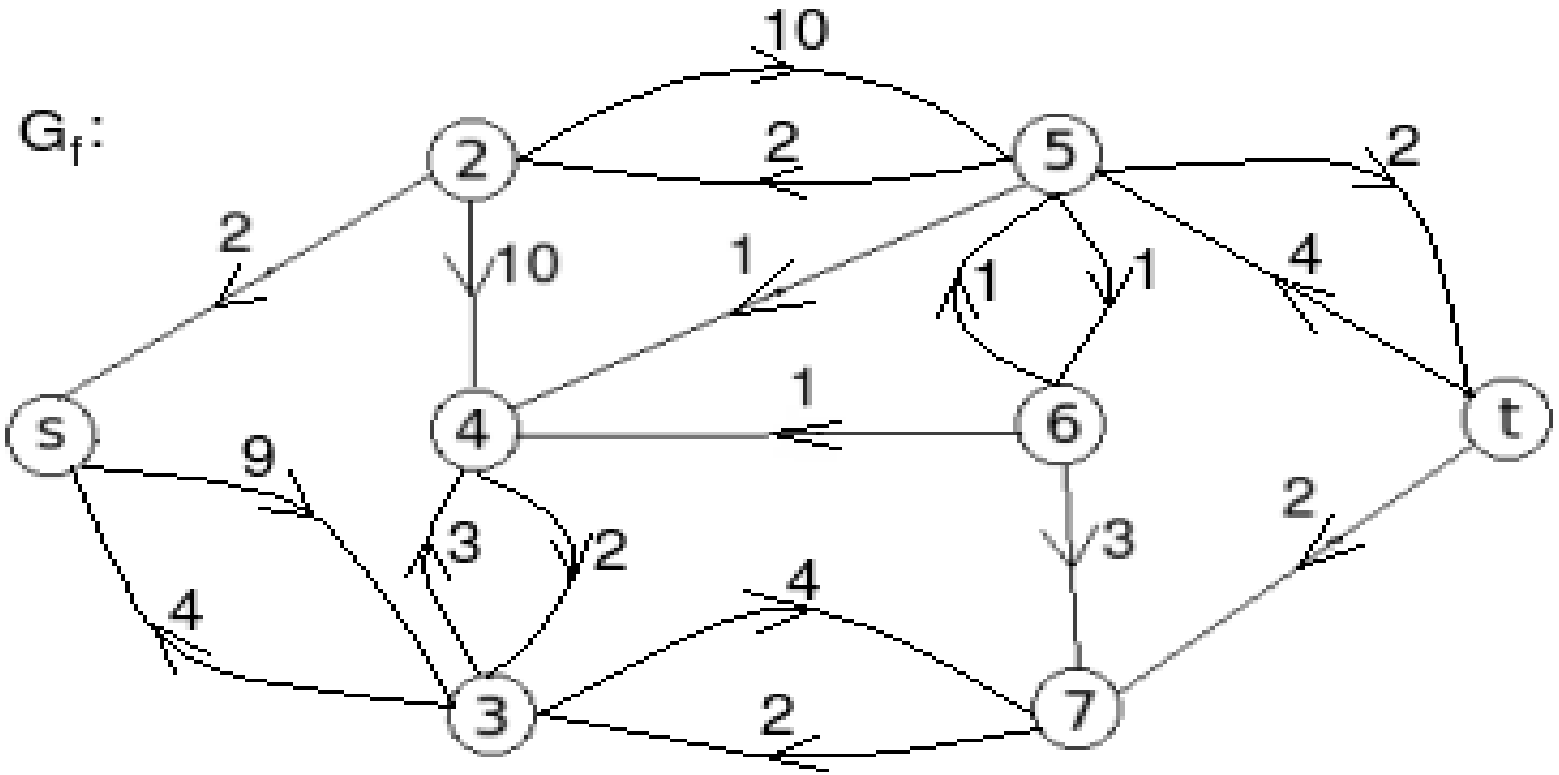
# Solution



**Exercise2:** Apply Ford-Fulkerson algorithm to find maximum flow in the following network.



# Solution:




Max flow=6