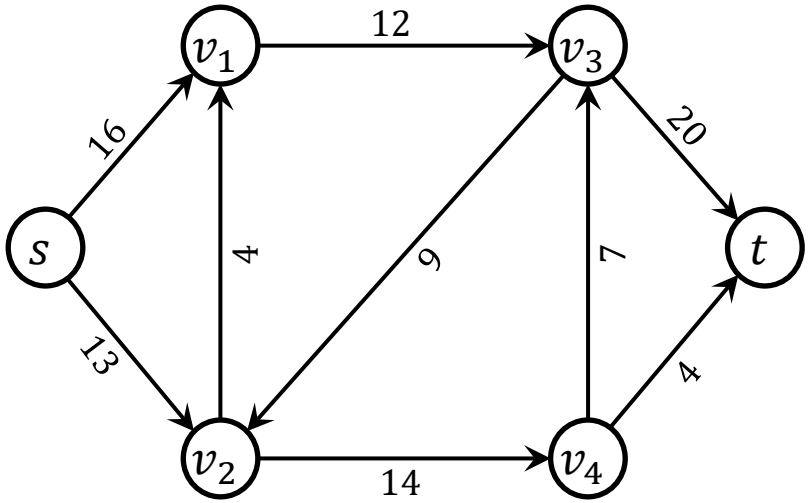
CSE 548 / AMS 542: Analysis of Algorithms

Prerequisites Review 9
(Network Flow
and Bipartite Matching)

Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023

Flow Networks

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$.



Also, if $(u, v) \in E$ then $(v, u) \notin E$.

If $(u, v) \notin E$, then we define $c(u, v) = 0$ for convenience.

We disallow self-loops.

We distinguish two vertices in a flow network: a ***source*** s and a ***sink*** t.

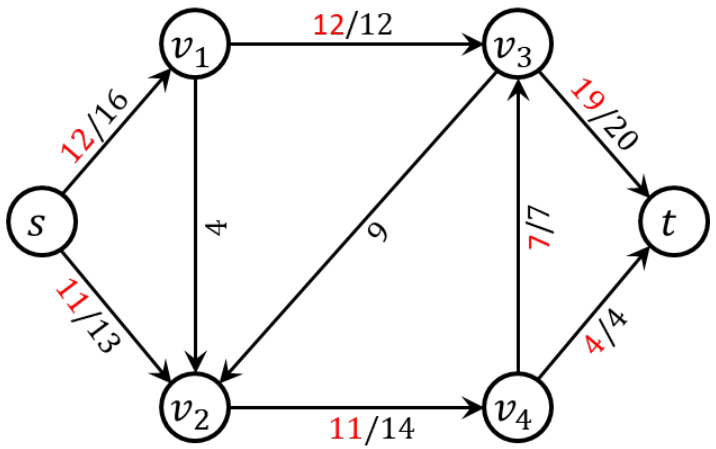
For convenience, we assume that each vertex lies on some path from s to t. The graph is therefore connected and, since each vertex other than s has at least one entering edge, $|E| \geq |V| - 1$.

Flows and the Maximum Flow Problem

Let $G = (V, E)$ be a flow network with a capacity function c.

Let s be the source and let t be the sink.

A ***flow*** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:



Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For all $u \in V \setminus \{s, t\}$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

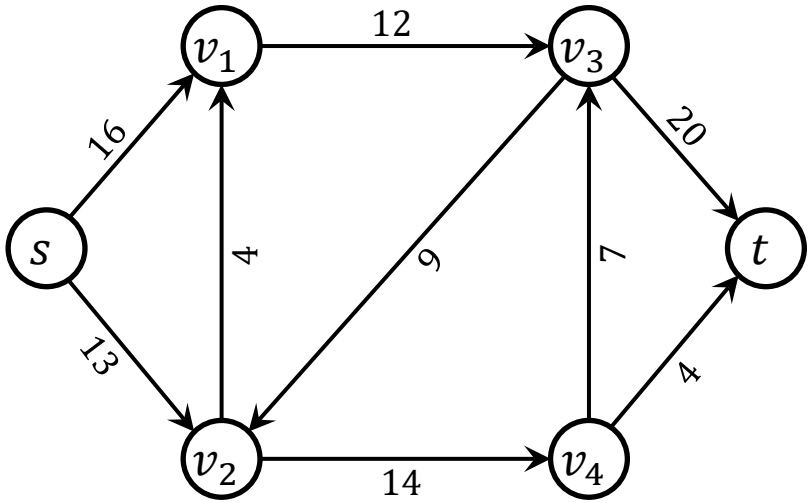
When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

The ***value*** $|f|$ of a flow f is defined as: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$.

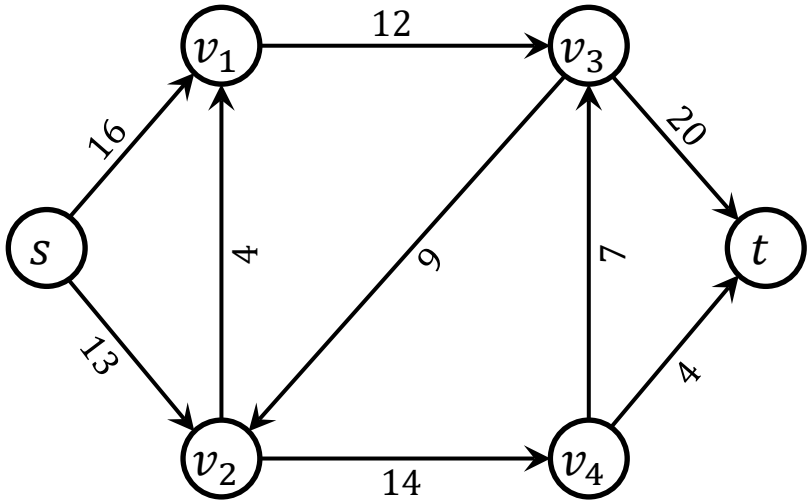
In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value.

Maximum Flow: The Ford-Fulkerson Method

Original Network

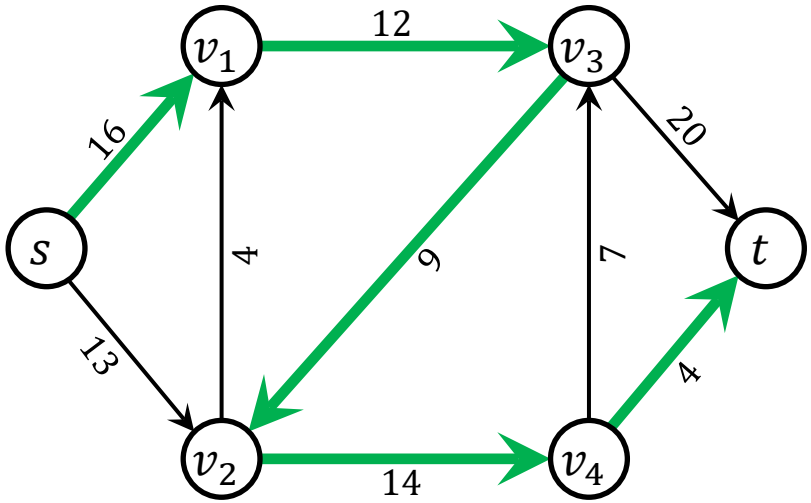


Original Network



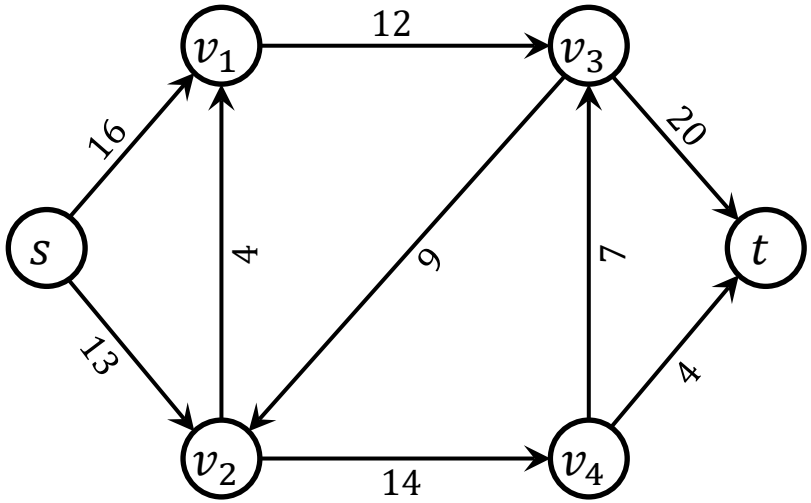
Maximum Flow: The Ford-Fulkerson Method

Step 1: Augmenting Path



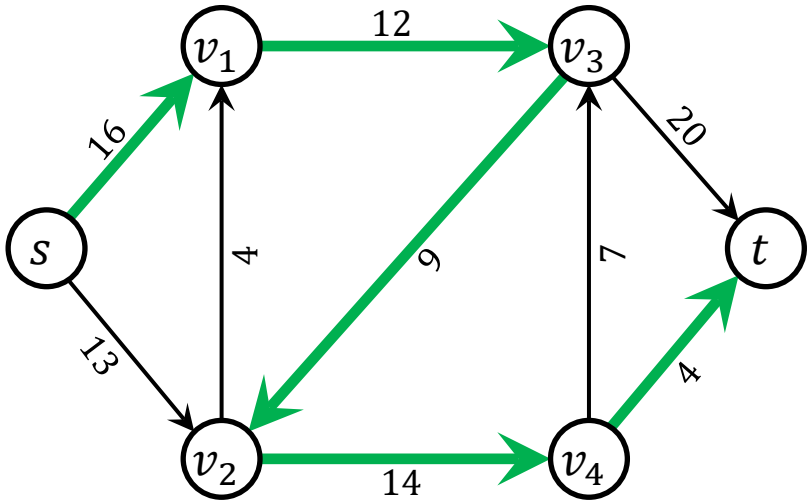
Residual capacity $= 4$

Original Network



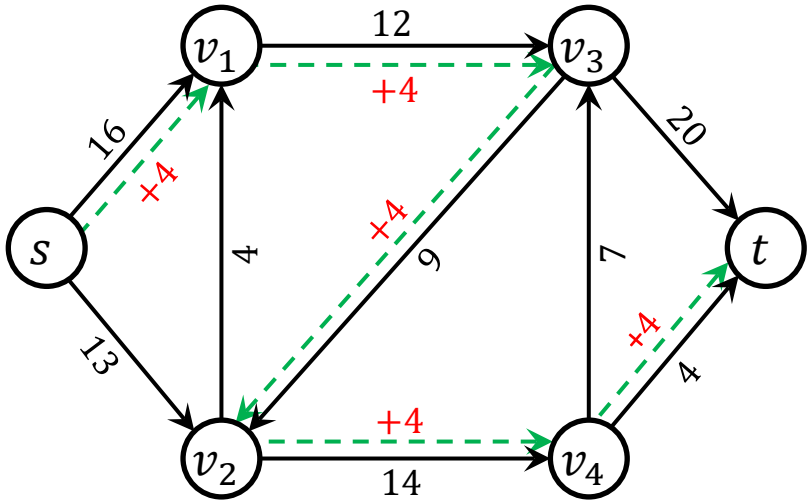
Maximum Flow: The Ford-Fulkerson Method

Step 1: Augmenting Path



Residual capacity $= 4$

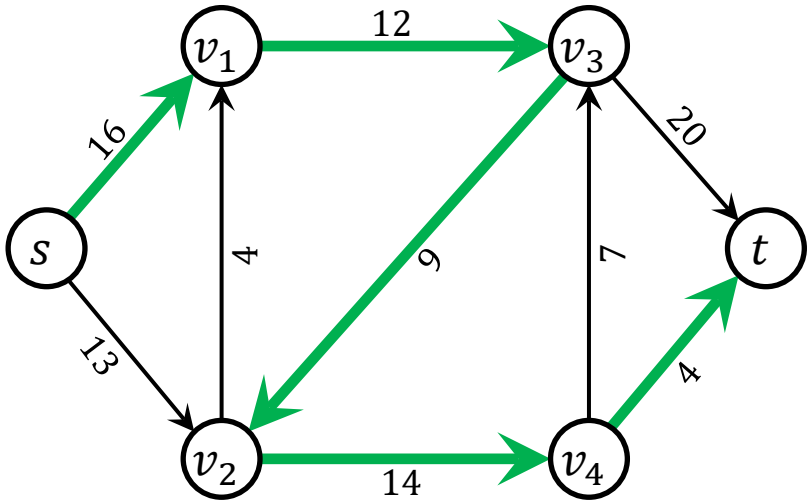
Step 1: Updating Flow



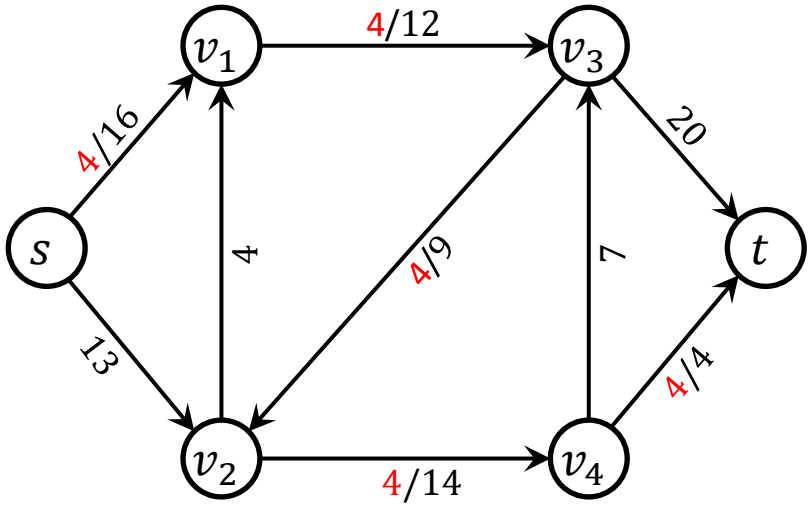
Increase flow by 4 along path
$s \rightarrow v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_4 \rightarrow t$

Maximum Flow: The Ford-Fulkerson Method

Step 1: Augmenting Path



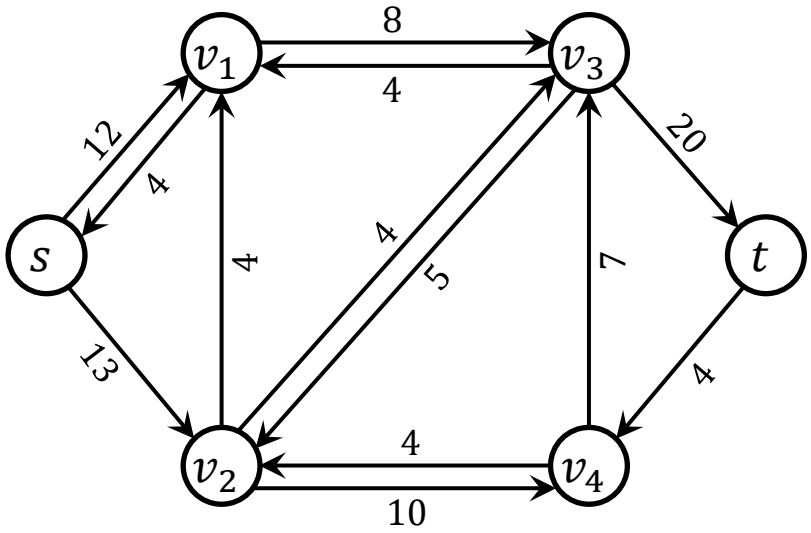
Step 1: Updated Flow



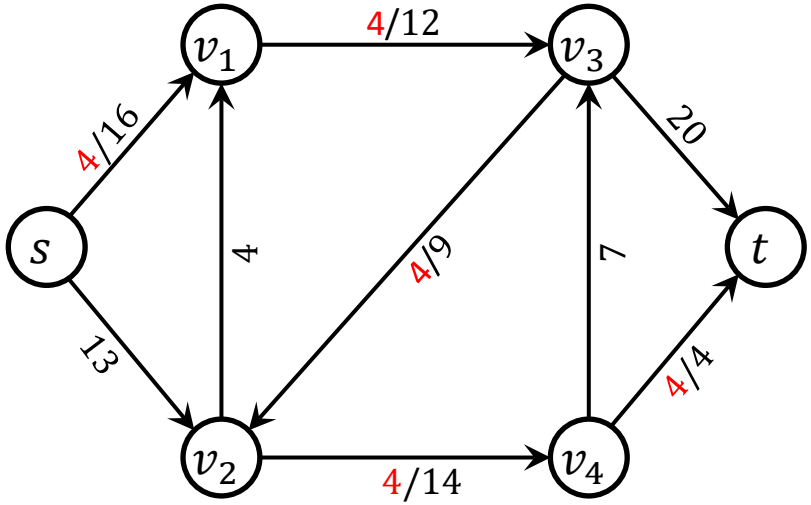
Current s to t flow $= 4$

Maximum Flow: The Ford-Fulkerson Method

Step 2: Residual Network

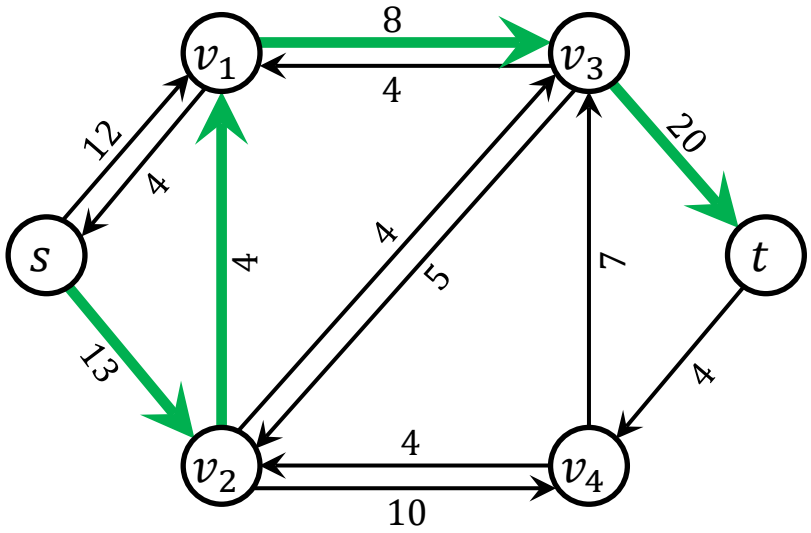


Step 1: Updated Flow



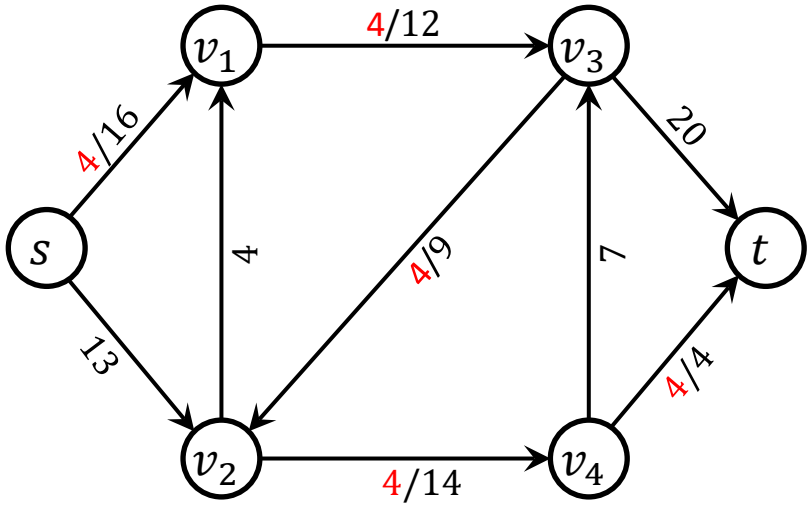
Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path



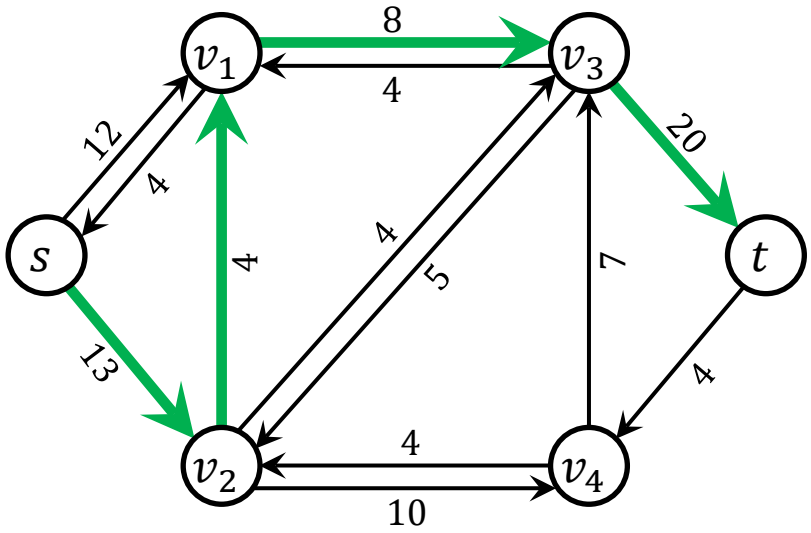
Residual capacity $= 4$

Step 1: Updated Flow



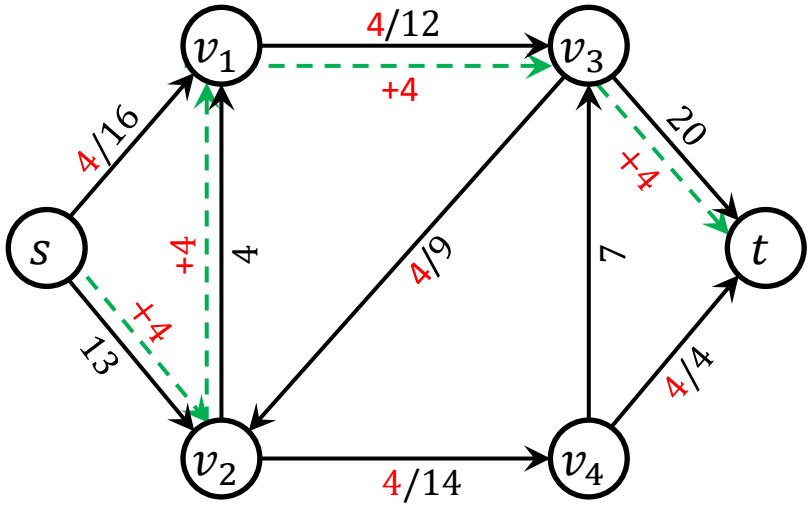
Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path



Residual capacity $= 4$

Step 2: Updating Flow

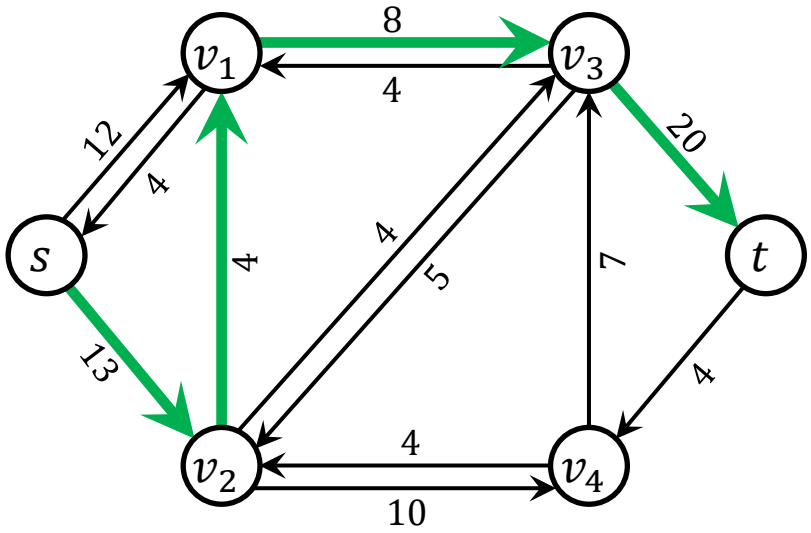


Increase flow by 4 along path

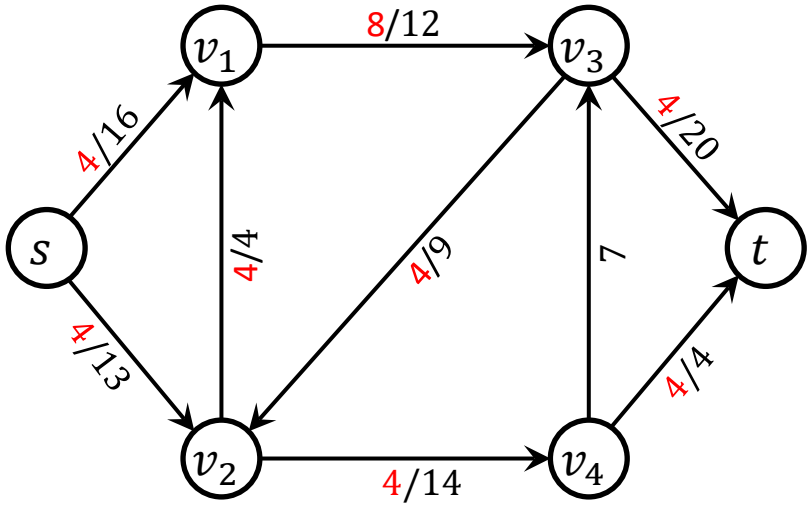
$s \rightarrow v_4 \rightarrow v_1 \rightarrow v_3 \rightarrow t$

Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path



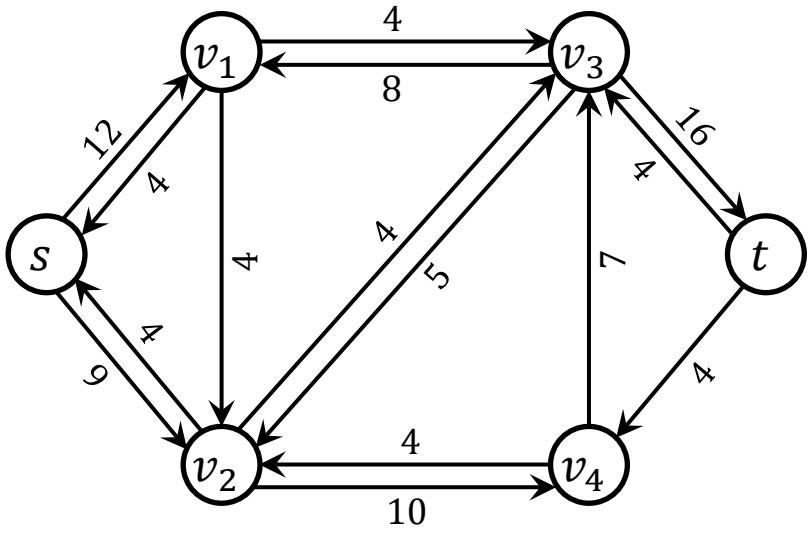
Step 2: Updated Flow



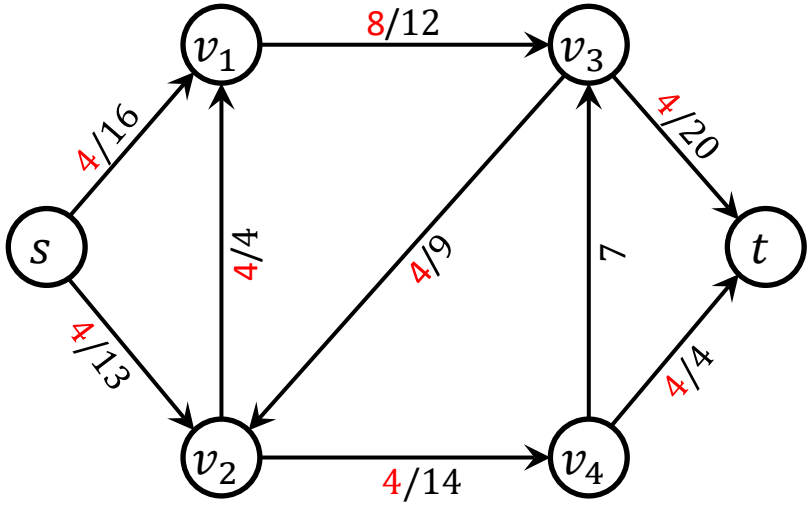
Current s to t flow $= 8$

Maximum Flow: The Ford-Fulkerson Method

Step 3: Residual Network

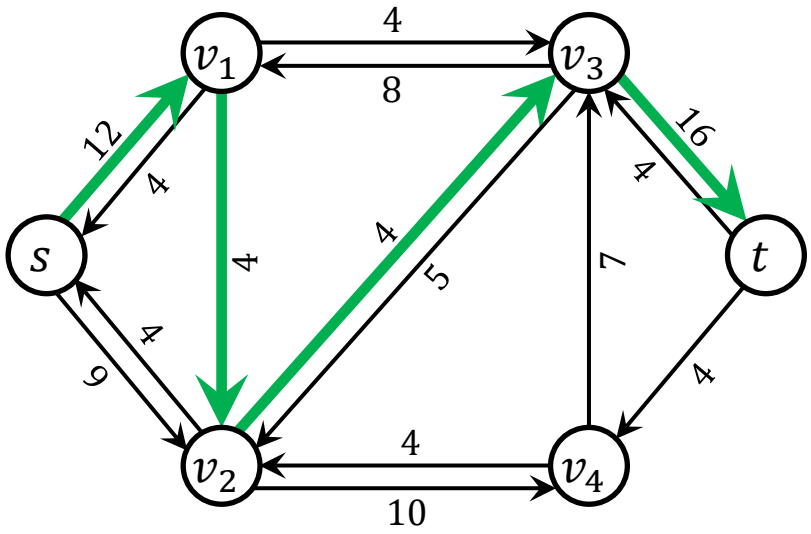


Step 2: Updated Flow



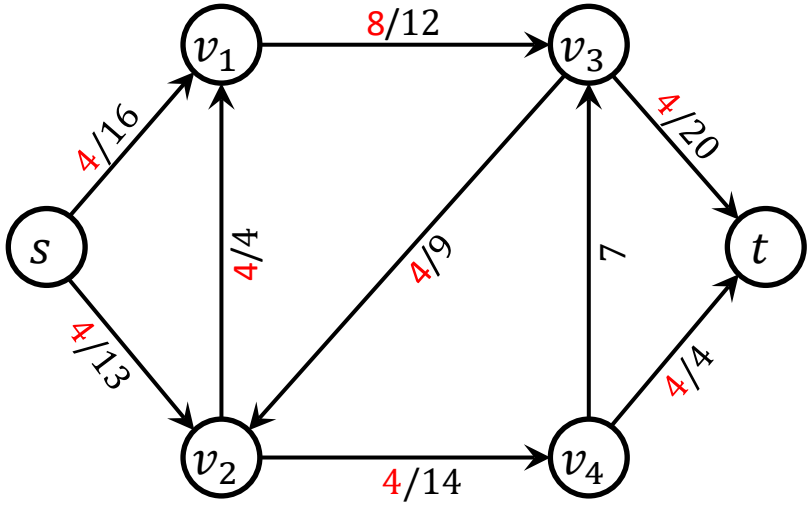
Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path



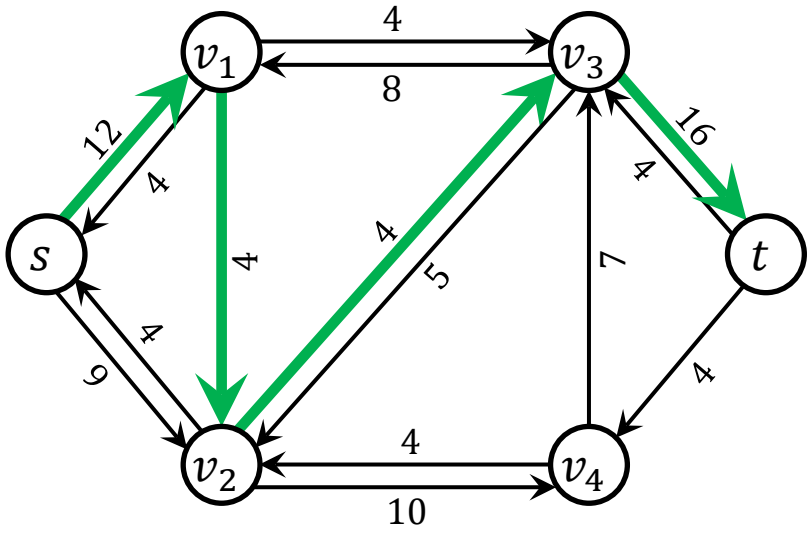
Residual capacity $= 4$

Step 2: Updated Flow



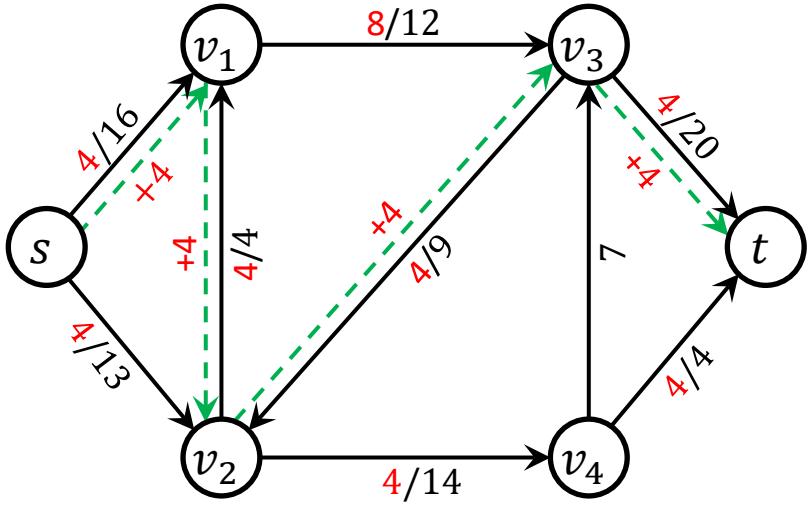
Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path



Residual capacity $= 4$

Step 3: Updating Flow

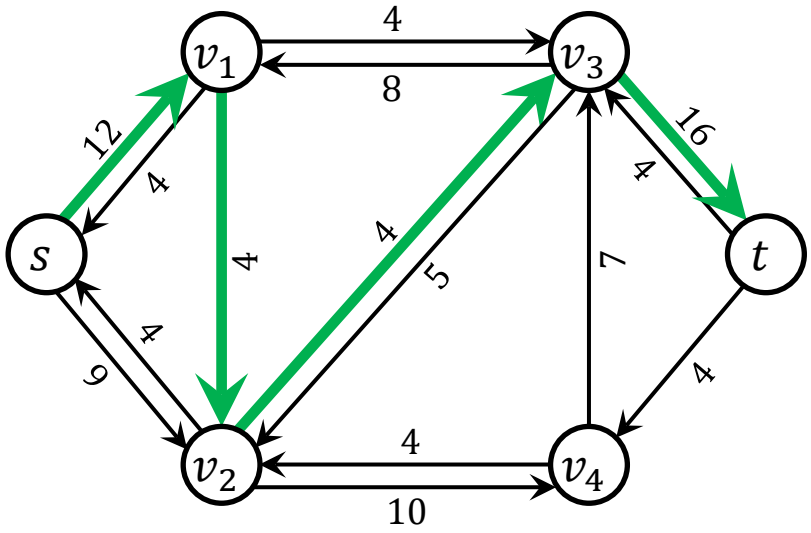


Increase flow by 4 along path

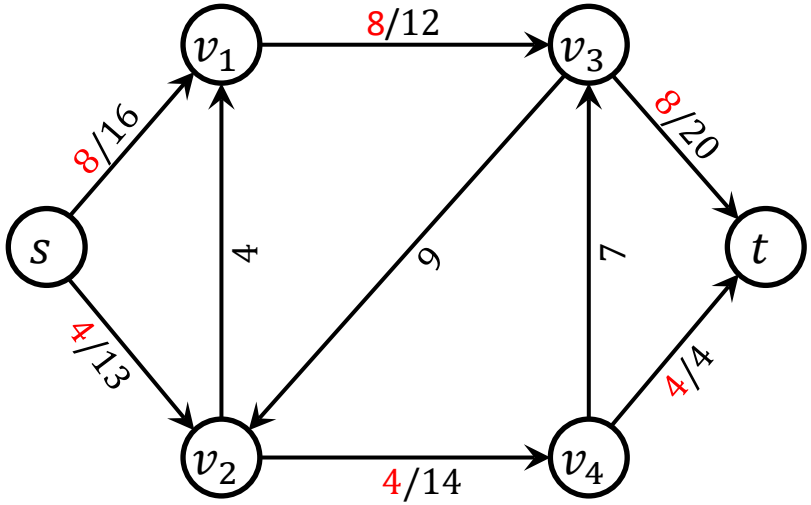
$s \rightarrow v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow t$

Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path



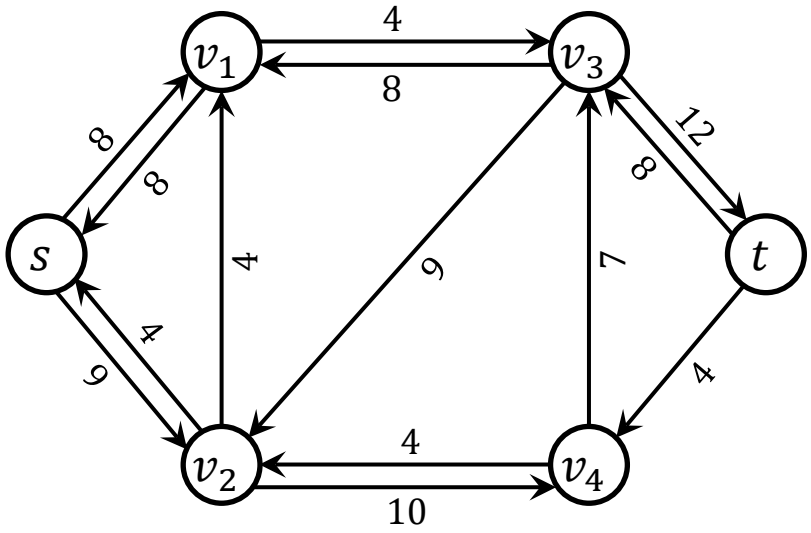
Step 3: Updated Flow



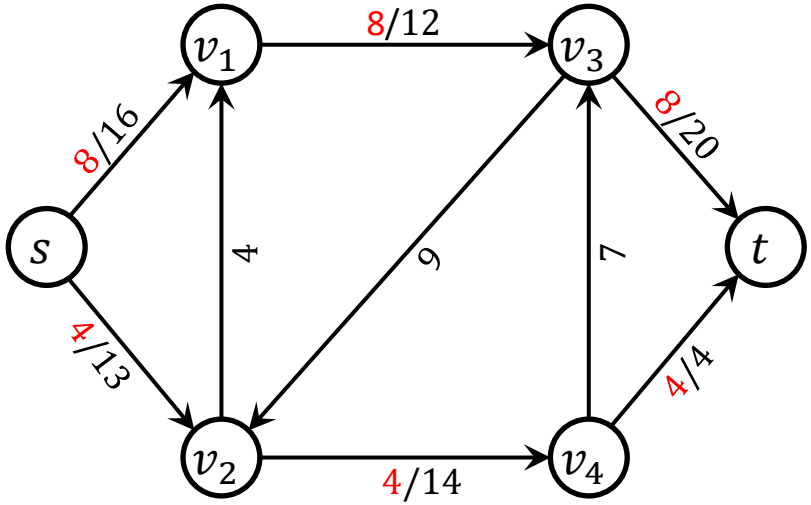
Current s to t flow $= 12$

Maximum Flow: The Ford-Fulkerson Method

Step 4: Residual Network

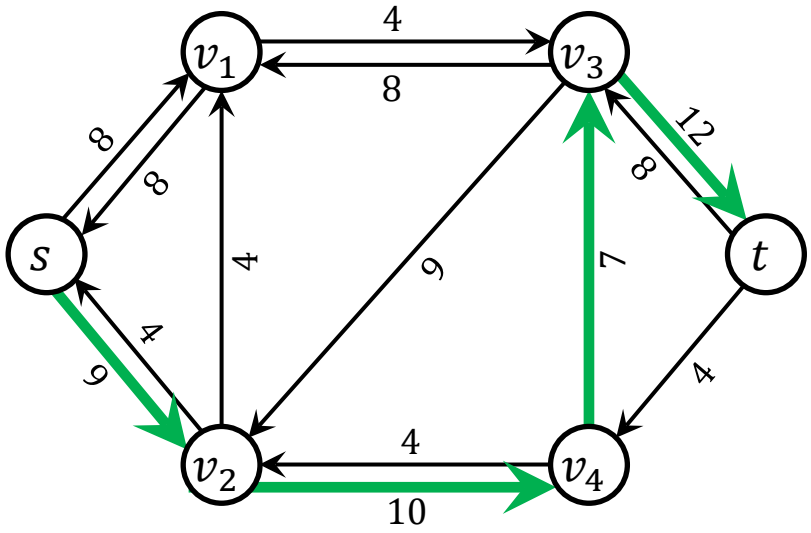


Step 3: Updated Flow



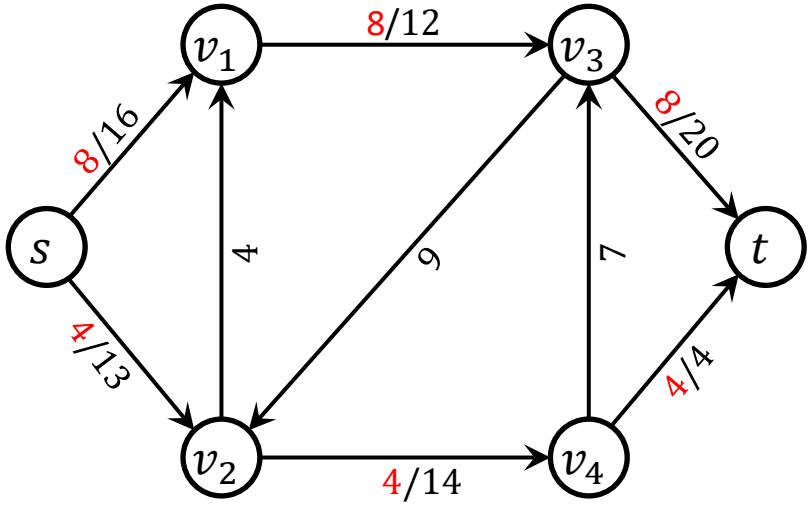
Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path



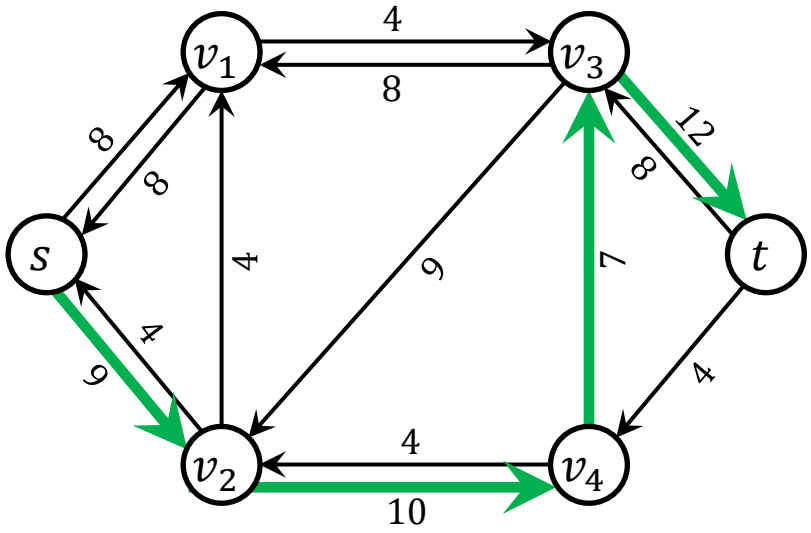
Residual capacity $= 7$

Step 3: Updated Flow

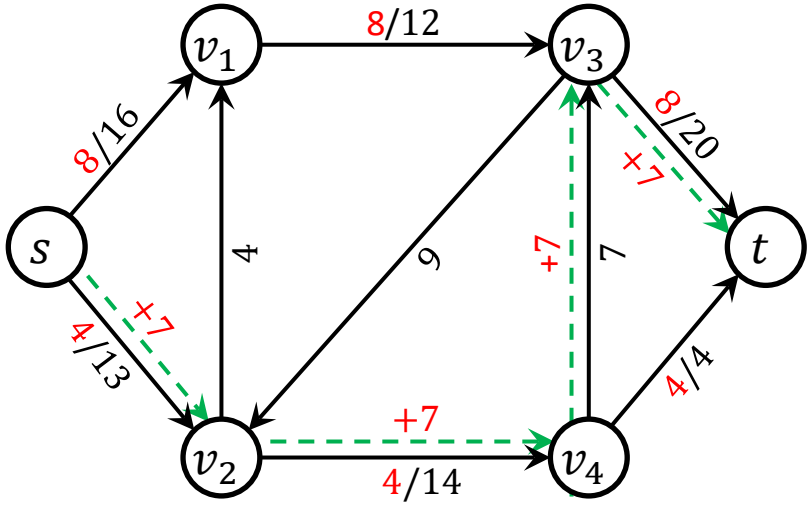


Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path



Step 4: Updating Flow

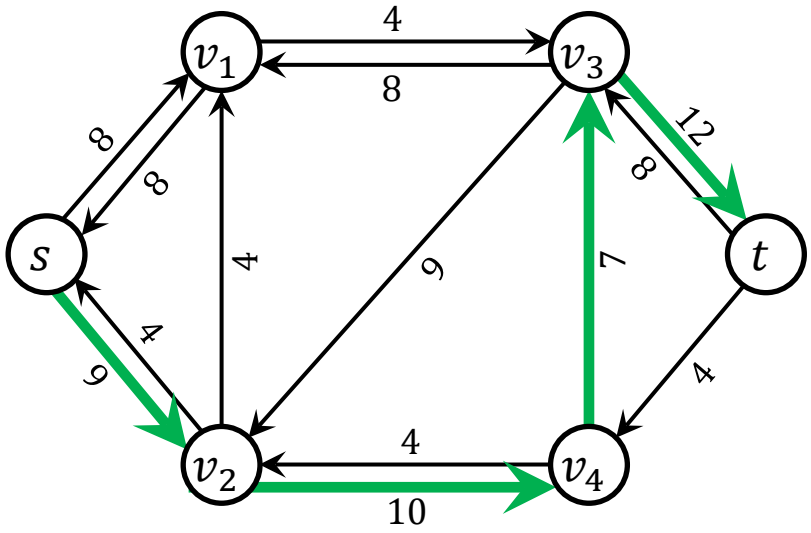


Increase flow by 7 along path

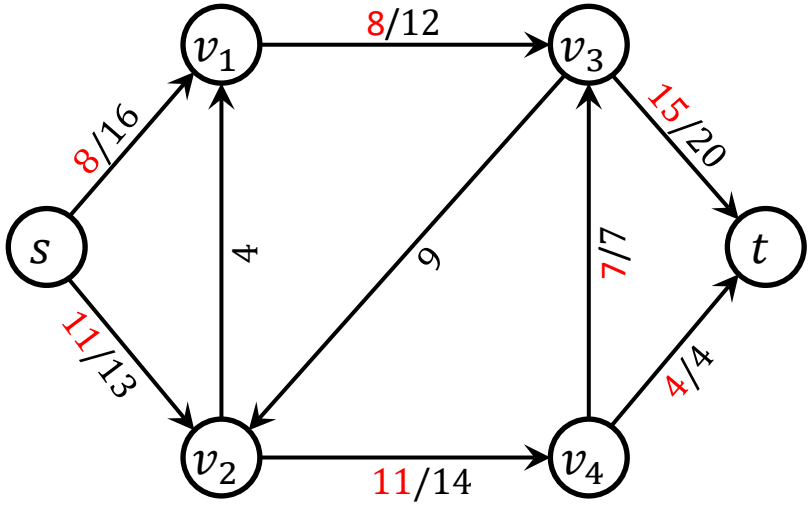
$$s \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow t$$

Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path



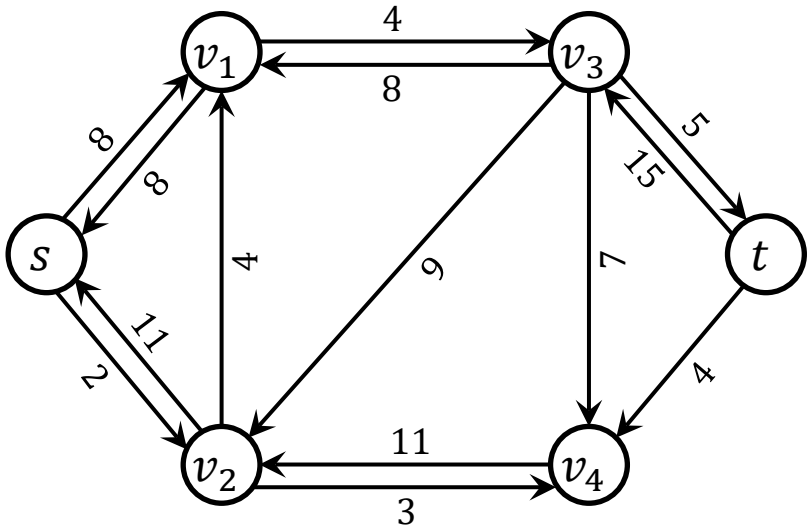
Step 4: Updated Flow



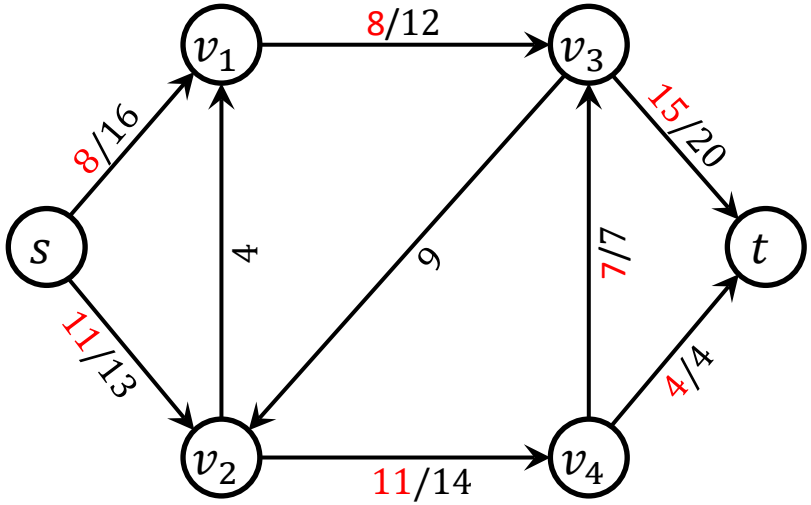
Current s to t flow $= 19$

Maximum Flow: The Ford-Fulkerson Method

Step 5: Residual Network

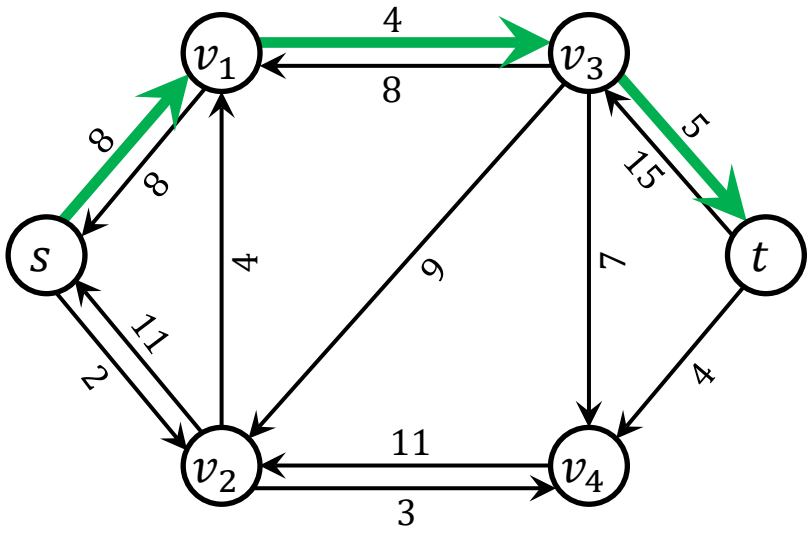


Step 4: Updated Flow



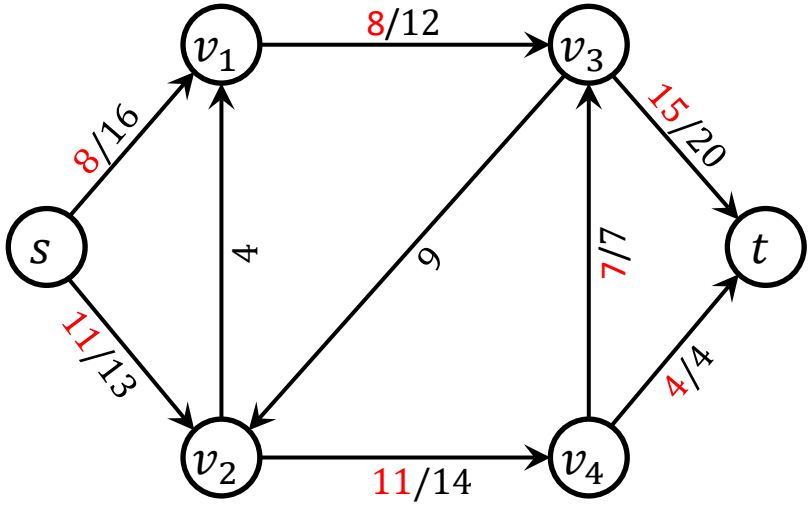
Maximum Flow: The Ford-Fulkerson Method

Step 5: Augmenting Path



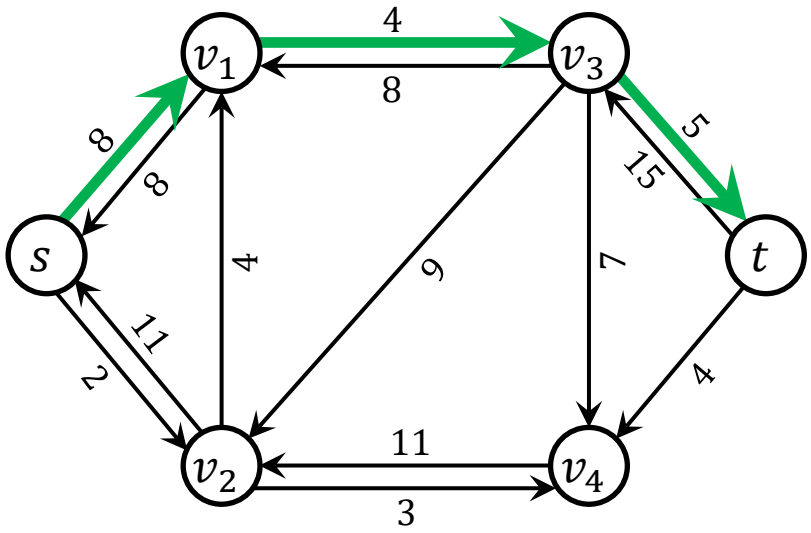
Residual capacity $= 4$

Step 4: Updated Flow



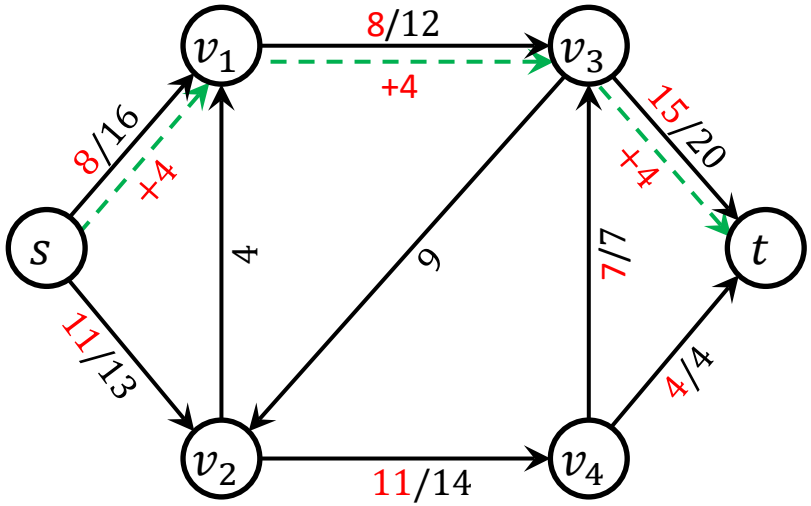
Maximum Flow: The Ford-Fulkerson Method

Step 5: Augmenting Path



Residual capacity $= 4$

Step 5: Updating Flow

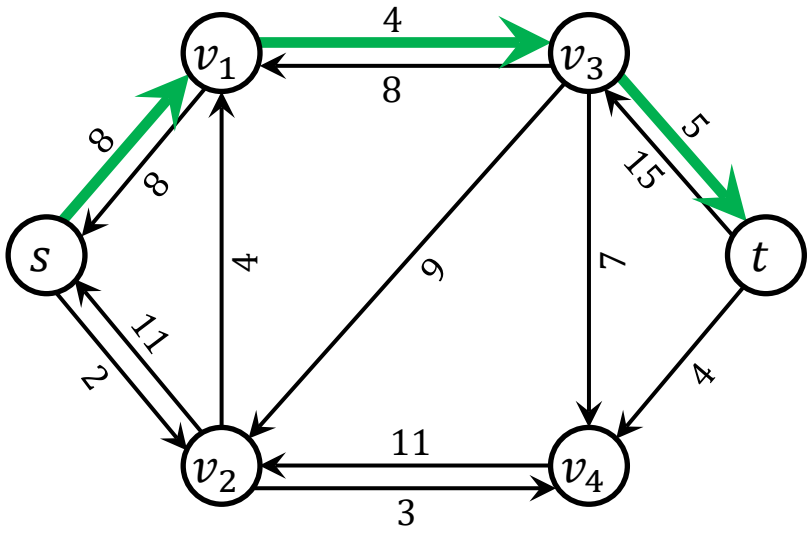


Increase flow by 4 along path

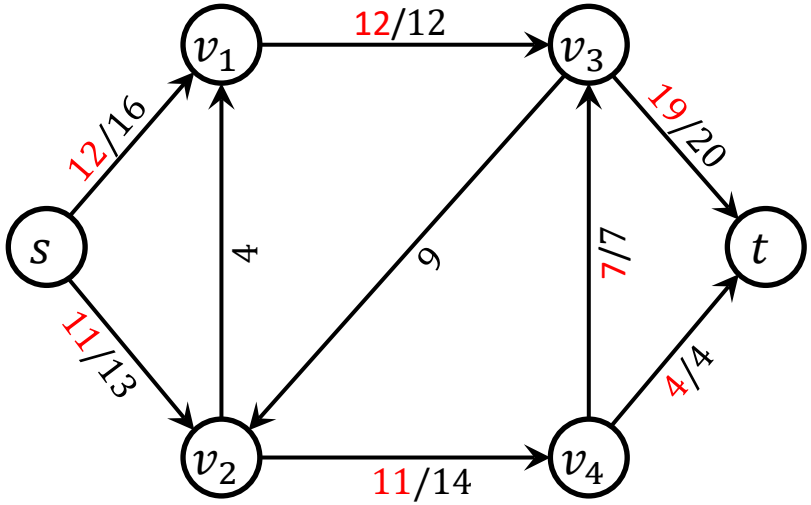
$s \rightarrow v_1 \rightarrow v_3 \rightarrow t$

Maximum Flow: The Ford-Fulkerson Method

Step 5: Augmenting Path



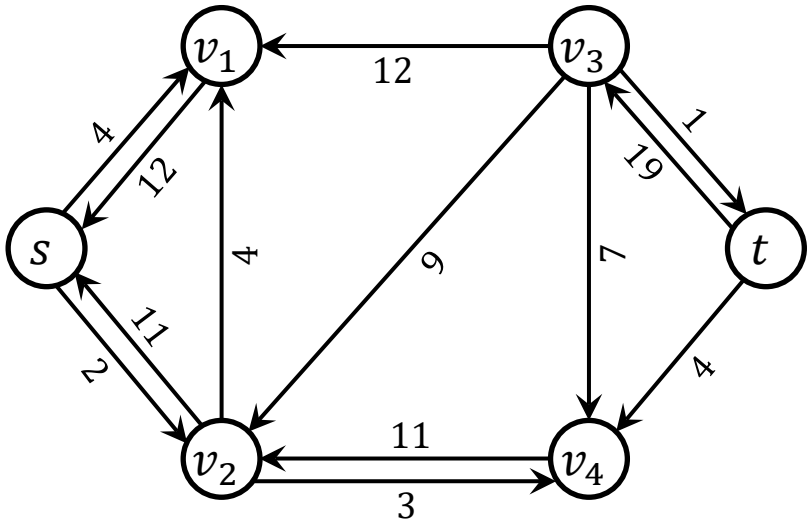
Step 5: Updated Flow



Current s to t flow $= 23$

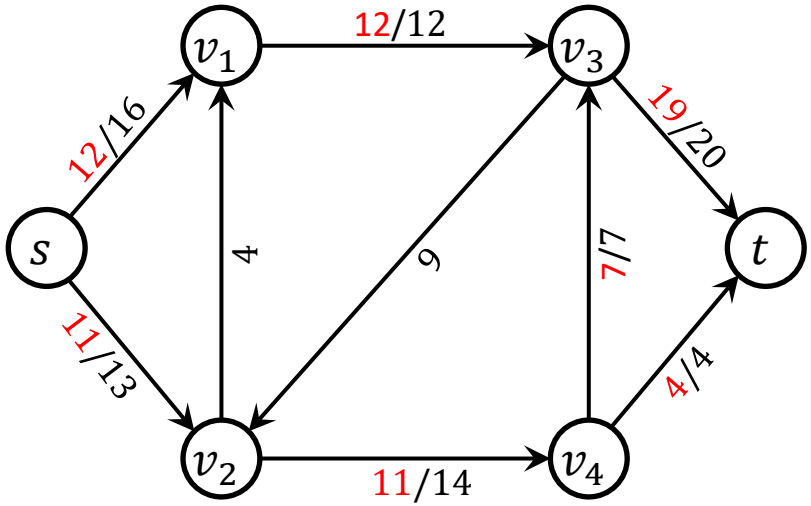
Maximum Flow: The Ford-Fulkerson Method

Step 6: Residual Network



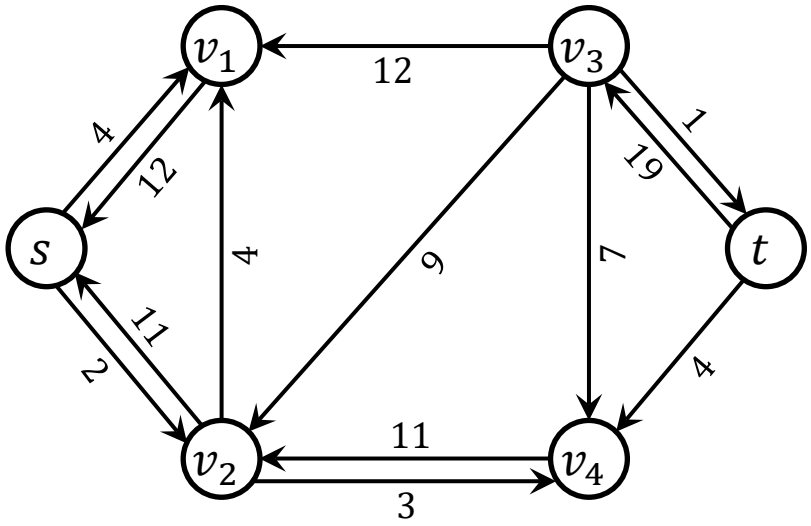
No augmenting path!

Step 5: Updated Flow



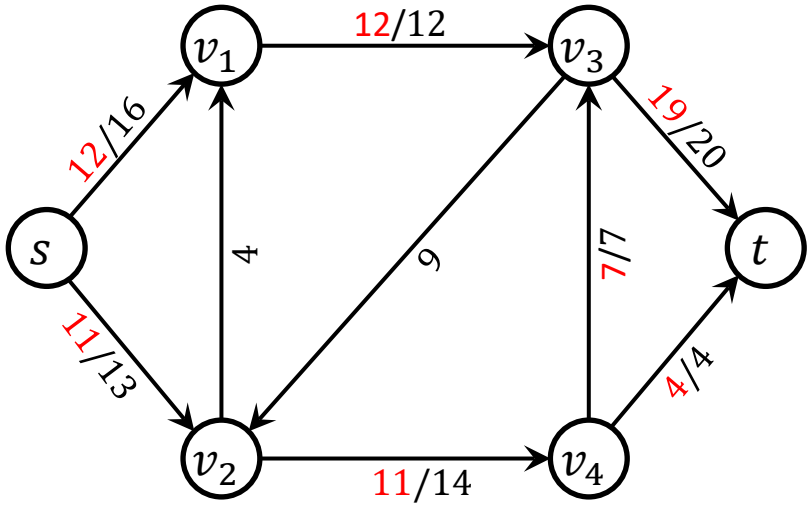
Maximum Flow: The Ford-Fulkerson Method

Step 6: Residual Network



No augmenting path!

Step 6: No Update



Done!

Maximum s to t flow $= 23$

Maximum Flow: The Ford-Fulkerson Method

Input: A flow network $G = (V, E)$ with a capacity function c, a source vertex s and a sink vertex t.

Output: A maximum s *to* t flow.

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

The Ford-Fulkerson Method: Running Time

The running time of *FORD-FULKERSON* depends on how we find the augmenting paths.

If we choose the augmenting paths poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value (e.g., might happen when the capacities are irrational numbers).

In practice, the capacities are often integral.

If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral.

If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of *FORD-FULKERSON* requires to find an augmenting path at most $|f^*|$ times, since each augmentation increases the flow value by at least one unit.

The Ford-Fulkerson Method: Running Time

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

Once the residual network G_f is known, an augmenting path can be found in $O(m+n)$ time using either a depth-first or a breadth-first search, where $n = |V|$ and $m = |E|$.

It is also easy to maintain the network, capacities and flows in a way that allows one to find G_f and update the flows in $O(m+n)$ time during each augmentation.

The running time of *FORD-FULKERSON* is thus $O\big((m+n)|f^*|\big)$ which is simply $O(m|f^*|)$ as $m = \Omega(n)$.

The Edmonds-Karp Algorithm

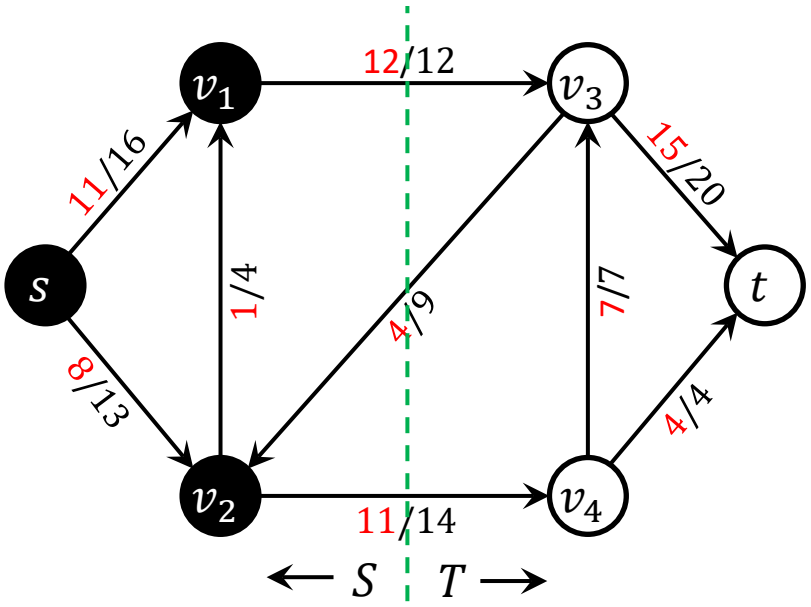
```
FORD-FULKERSON ( G = (V, E), s, t )
1.    for each edge (u, v) ∈ G.E do
2.        (u, v).f ← 0
3.    while there exists a path p from s to t in the residual network G_f do
4.        c_f(p) ← min{c_f(u, v) | (u, v) is in p}
5.        for each edge (u, v) in p do
6.            if (u, v) ∈ G.E then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

The Edmonds-Karp algorithm is an implementation of the *FORD-FULKERSON* method in which the augmenting path p in line 3 is found using a breadth-first search.

That is, p is chosen as a shortest path from s to t in the residual network, where each edge has unit distance (weight).

One can show that the Edmonds-Karp algorithm runs in $O(m^2 n)$ time, where $n = |G.V|$ and $m = |G.E|$.

Cuts of Flow Networks (Max-flow Min-cut Theorem)



A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V \setminus S$ such that $s \in S$ and $t \in T$.

(Unlike the "cut" used for MST's, here the graph is directed, and we insist that $s \in S$ and $t \in T$.)

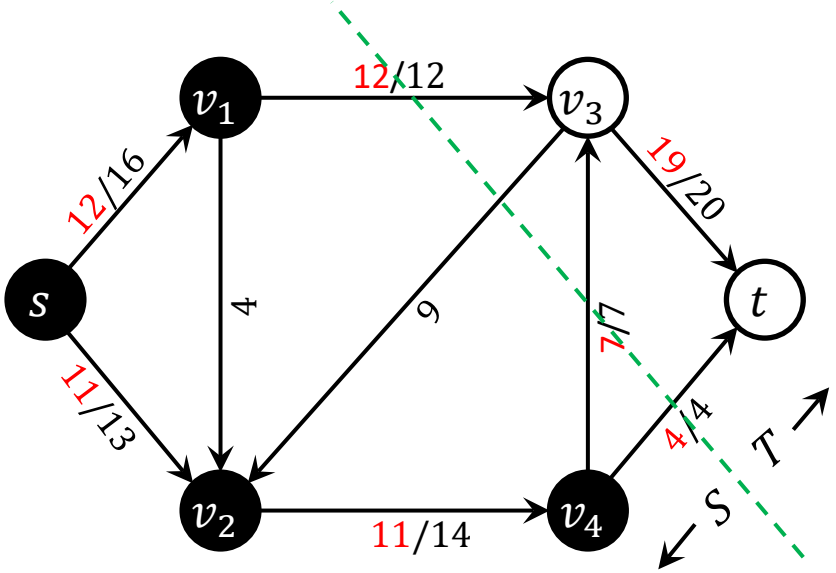
If f is a flow, then the ***net flow*** $f(S, T)$ across (S, T) is defined to be

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

The capacity of the cut (S, T) is: $c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$.

A ***minimum cut*** of a network is a cut of minimum capacity.

Cuts of Flow Networks (Max-flow Min-cut Theorem)



THEOREM (CLRS): If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

The theorem above says that, in a flow network:

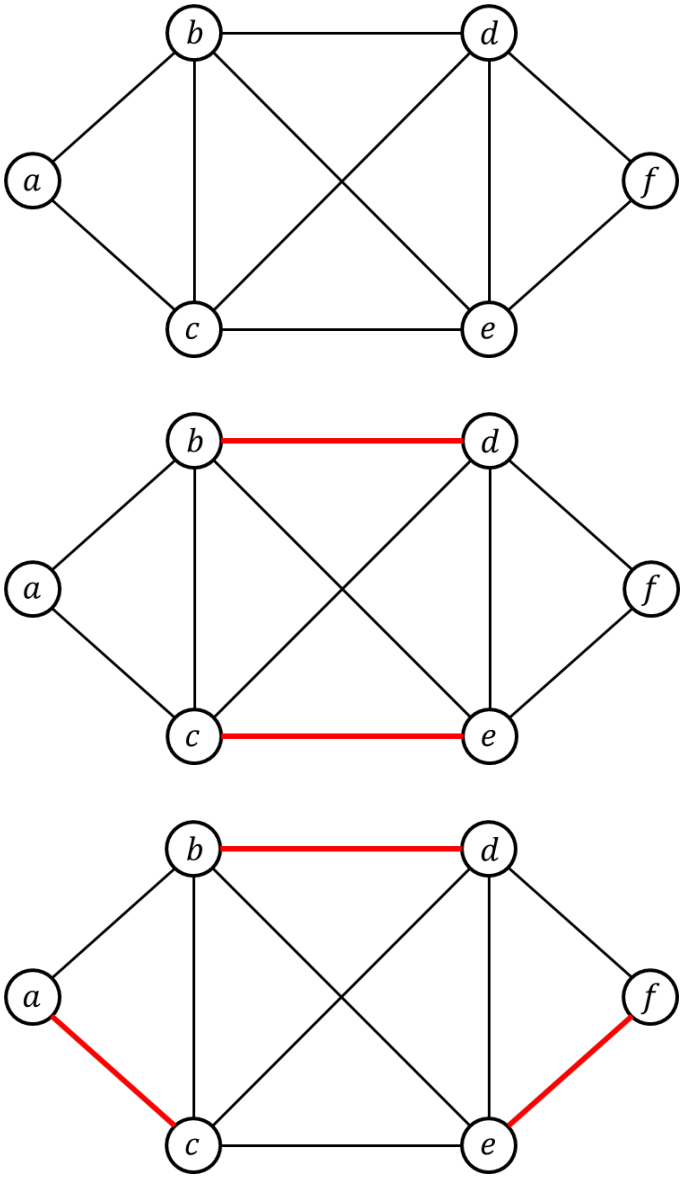
$$value\ of\ a\ maximum\ flow = capacity\ of\ a\ minimum\ cut$$

The Maximum Matching Problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.

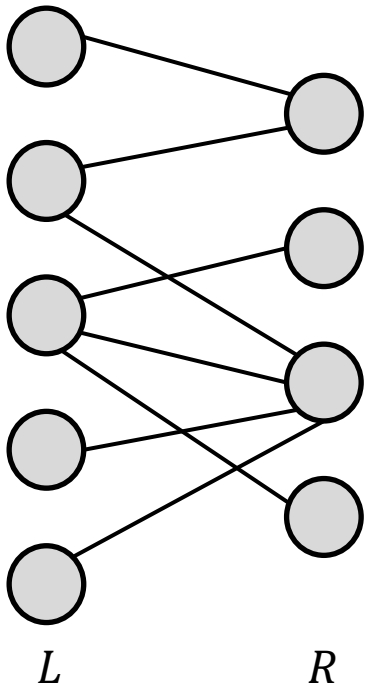
We say that a vertex $v \in V$ is matched by the matching M if some edge in M is incident on v; otherwise, v is unmatched.

A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$.



The Maximum Bipartite Matching Problem

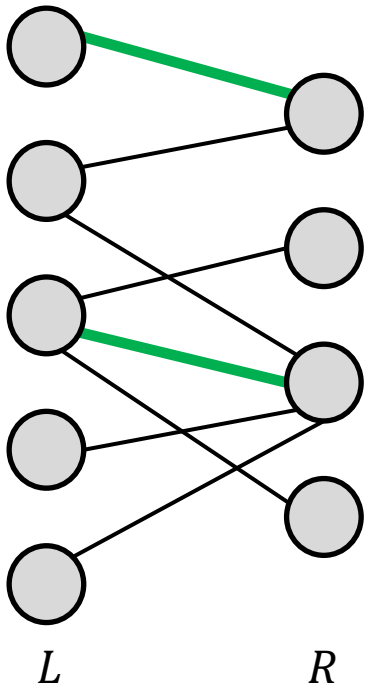
We shall restrict our attention to finding ***maximum matchings in bipartite graphs***: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.



A bipartite graph

The Maximum Bipartite Matching Problem

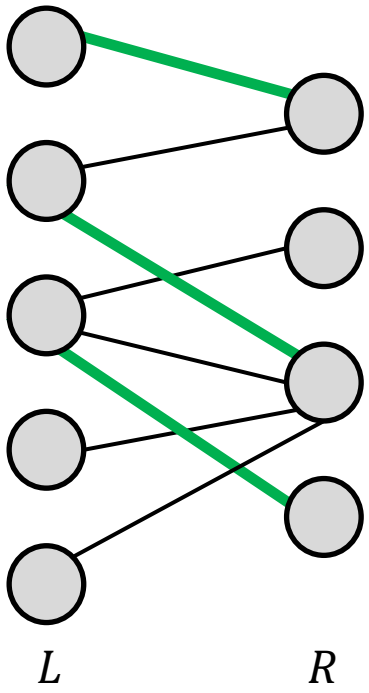
We shall restrict our attention to finding ***maximum matchings in bipartite graphs***: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.



A bipartite matching

The Maximum Bipartite Matching Problem

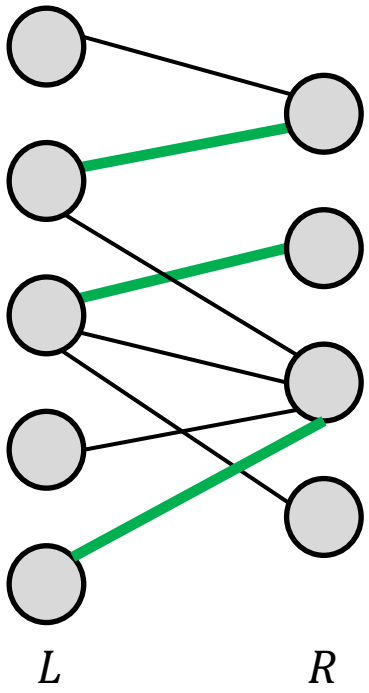
We shall restrict our attention to finding ***maximum matchings in bipartite graphs***: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.



A maximum bipartite matching

The Maximum Bipartite Matching Problem

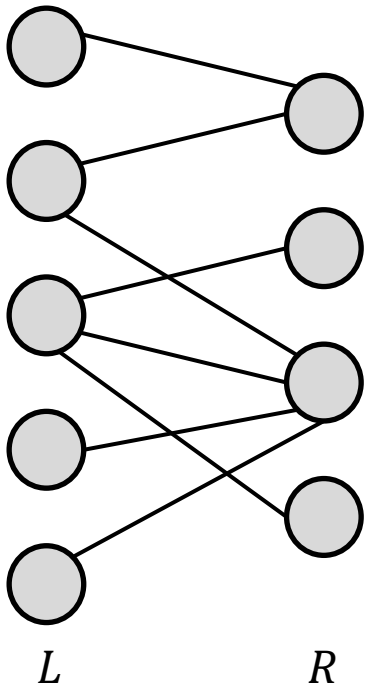
We shall restrict our attention to finding ***maximum matchings in bipartite graphs***: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.



Another maximum bipartite matching

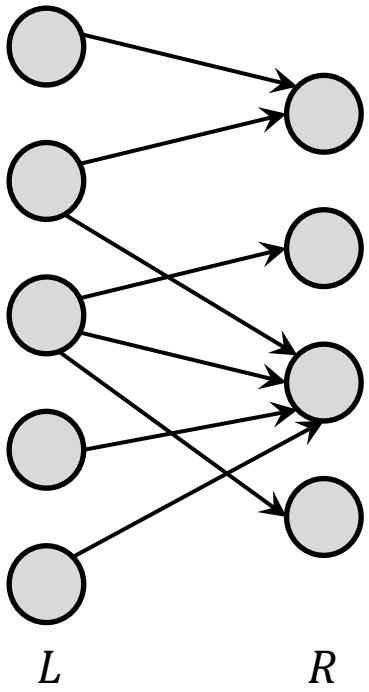
Maximum Bipartite Matching using Network Flow

Given a bipartite graph $G = (V, E)$ with $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R.



Maximum Bipartite Matching using Network Flow

First, direct all edges from L to R.

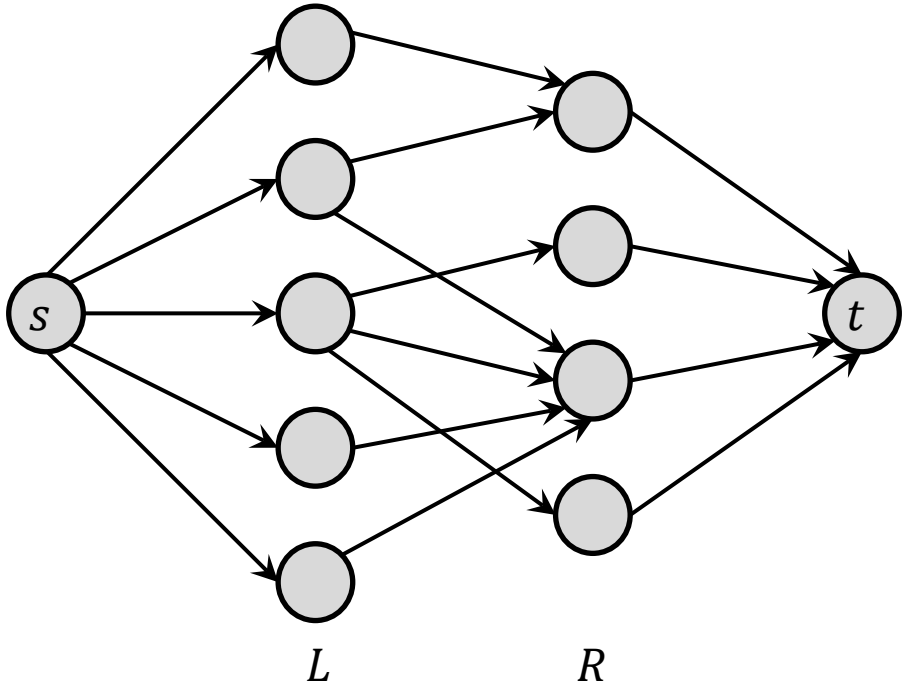


Maximum Bipartite Matching using Network Flow

Then add a source s and a sink t.

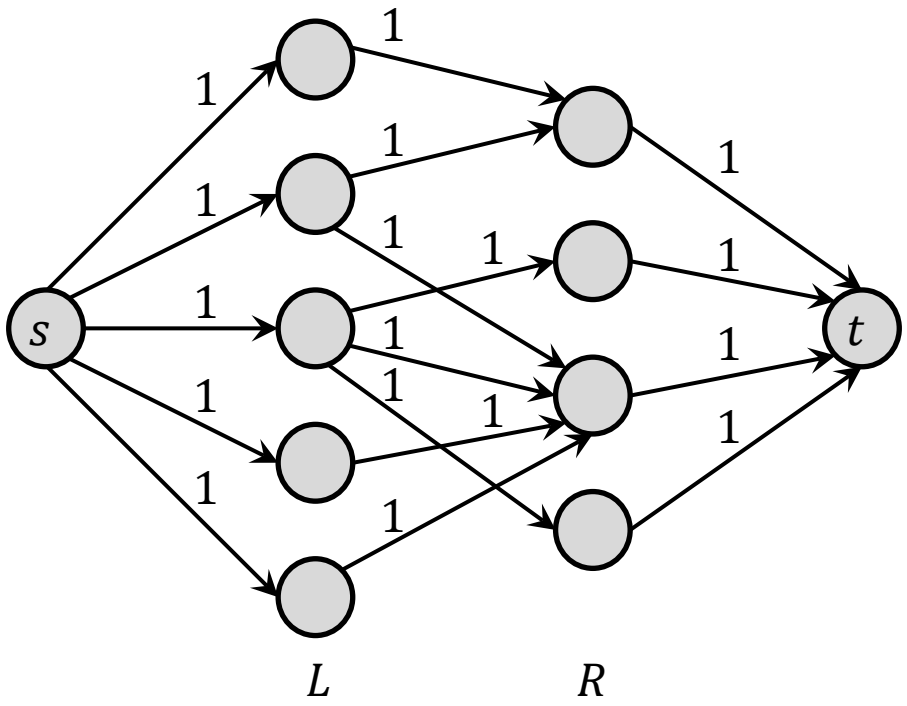
For every vertex $v \in L$, add an edge (s, v) directed from s to v.

For every vertex $v \in R$, add an edge (v, t) directed from v to t.



Maximum Bipartite Matching using Network Flow

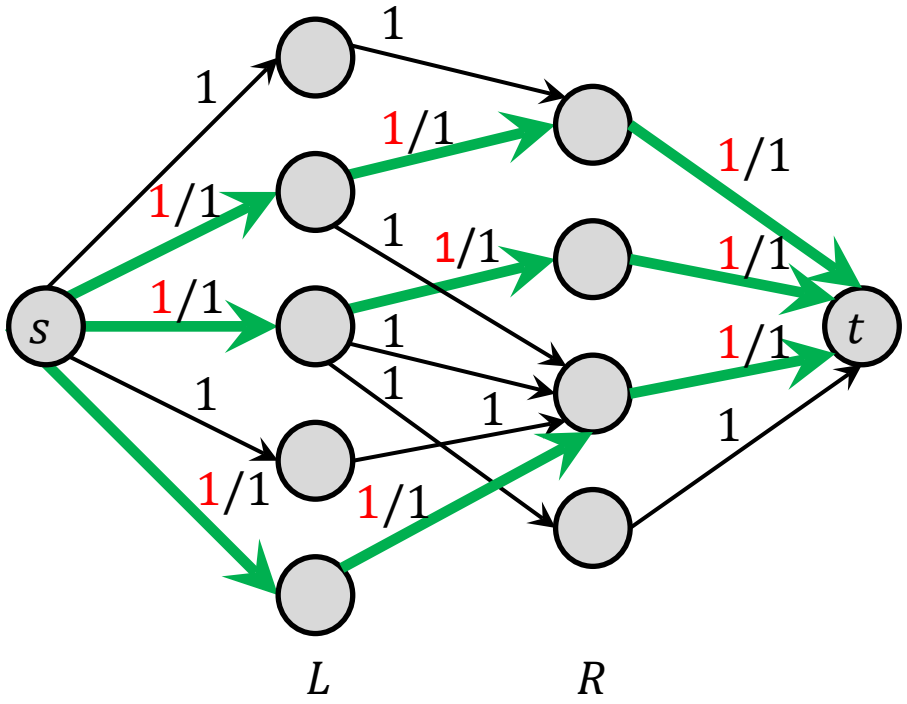
For every edge (u, v) in this new directed graph, set capacity $c(u, v) = 1$.



Maximum Bipartite Matching using Network Flow

Now, find a maximum s to t flow f^* in this new flow network G' using the *FORD-FULKERSON* method.

One can show that $|f^*|$ will always be an integer and will be equal to the maximum matching in the original bipartite graph.



Since $|f^*| < n = |L \cup R|$, running time will be $O(mn)$, where $m = |E|$.

The Flow-based Bipartite Matching Algorithm Works because of the Integrality Theorem

THEOREM (CLRS): If the capacity function c takes on only integer values, then the maximum flow f produced by the *FORD-FULKERSON* method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

COROLLARY (CLRS): The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Optional

Proof of Running Time of the Edmonds-Karp Algorithm

The Edmonds-Karp Algorithm: Running Time

LEMMA 26.7 (CLRS): If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and and sink t, then for all vertices $v \in G.V \setminus \{s, t\}$, the shortest path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

PROOF: Let's assume for contradiction that for some vertex $v \in G.V \setminus \{s, t\}$, there is a flow augmentation that causes the shortest-path distance from s to v to decrease.

Let f be the flow just before the first augmentation that decreases some shortest-path distance and let f' be the flow just afterward.

Let v be the vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$.

The Edmonds-Karp Algorithm: Running Time

PROOF (CONTINUED): Let $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v in G_f, so that $(u, v) \in E_f$, and $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.

Because of the way we chose v, we know that $\delta_{f'}(s, u) \geq \delta_f(s, u)$.

Then we must have $(u, v) \notin E_f$, as otherwise by triangle inequality:

$$\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v),$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.

How can we have $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$? The augmentation must have increased the flow from v to u. The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from s to u in G_f has (v, u) as its last edge.

Therefore, $\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$, which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.

The Edmonds-Karp Algorithm: Running Time

THEOREM 26.8 (CLRS): If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(mn)$, where $n = |G.V|$ and $m = |G.E|$.

PROOF: We say that an edge (u, v) in a residual network G_f is ***critical*** on an augmenting path p if $c_f(p) = c_f(u, v)$.

After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical.

We will show that each of the m edges can become critical at most $n/2$ times.

The Edmonds-Karp Algorithm: Running Time

PROOF (CONTINUED): Let $u, v \in G.V$ be connected by $(u, v) \in G.E$.

Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have $\delta_f(s, v) = \delta_f(s, u) + 1$.

Once the flow is augmented, the edge (u, v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v, u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$.

Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$ by Lemma 26.7, we have:

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2.$$

The Edmonds-Karp Algorithm: Running Time

PROOF (CONTINUED): Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from s increases by at least 2. The distance of u from s is initially at least 0. The intermediate vertices on a shortest path from s to u cannot contain s, u, or t (since (u, v) on an augmenting path implies that $u \neq t$). Therefore, until u becomes unreachable from s, if ever, its distance is at most $n - 2$. Thus, after the first time that (u, v) becomes critical, it can become critical at most$(n - 2)/2 = n/2 - 1$ times more, for a total of at most $n/2$ times.

Since there are $O(m)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the algorithm is $O(mn)$. Each augmenting path has at least one critical edge, and hence the theorem follows.