CS483: Analysis of Algorithms
Flow Networks

Prof. Ivan Avramovic, GMU

Max Flow

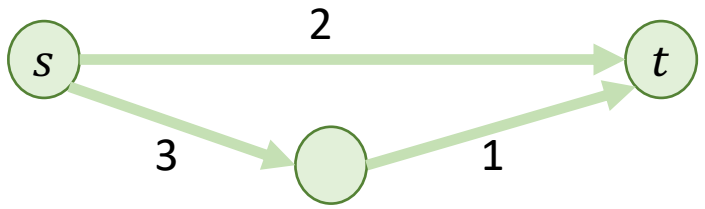
Imagine that in a weighted graph, our edge weights are **capacities**.

The capacities indicate **how much of something** can pass across an edge.

- **Example**: Amount of traffic.
- **Example**: Amount of network data.
- **Example**: Amount of liquid.

Goal of a *max flow* problem:

Find how much flow can be sent across the network without violating capacities.



Max Flow

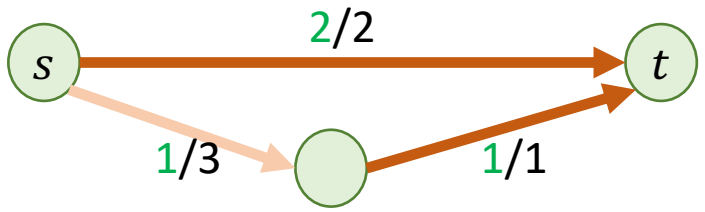
Imagine that in a weighted graph, our edge weights are **capacities**.

The capacities indicate **how much of something** can pass across an edge.

- **Example**: Amount of traffic.
- **Example**: Amount of network data.
- **Example**: Amount of liquid.

Goal of a *max flow* problem:

Find how much flow can be sent across the network without violating capacities.



A flow of 2 along the upper path and 1 along the lower path gives a total flow of $2 + 1 = 3$.

Flow Definitions

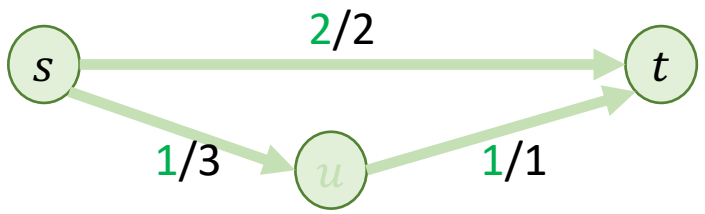
In a **flow network** for a directed graph $G = (V, E)$:

- Node $\boldsymbol{s}$ is the *source node* and $\boldsymbol{t}$ is the *sink node*.

  The flow originates from node $\boldsymbol{s}$ and flows into node $\boldsymbol{t}$.

- For any edge $e \in E$, its **capacity** is given by c_e.
- An *s-t flow* is a function f **defined over the edges** of the graph.

  Intuitively, a flow represents how much "stuff" is flowing across the network on that particular edge.



In this example, $c_{(s,u)} = 3$ and $f\big((s,u)\big) = 1$.

Max Flow, Objective

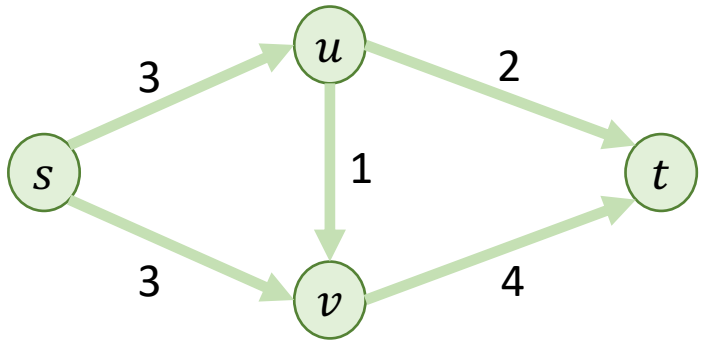
The *value* of a flow: total flow **leaving the source** or **entering the sink.**

Value of the flow $= \sum_{e \in (all\ edges\ leaving\ s)} f(e) = \sum_{e \in (all\ edges\ entering\ t)} f(e).$

Note that this will be the **same number** either way!

A max flow problem tries to **maximize** the value subject to constraints.

Assume that **all capacities are positive integers**. This is a very hard problem to solve if not.



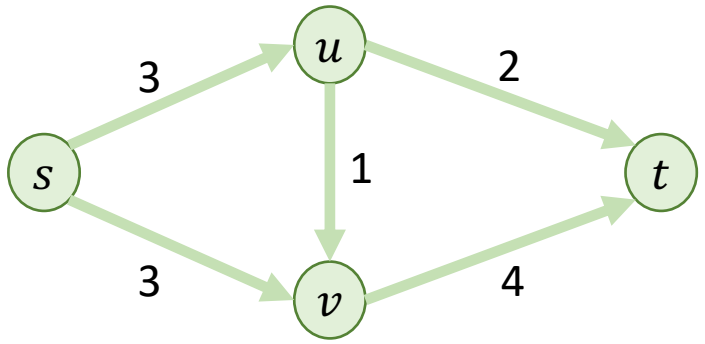
Max Flow, Constraints

Flow is **conserved** at each node other than s and t.

$$\sum_{e \in (edges\ entering\ u)} f(e) = \sum_{e \in (edges\ leaving\ u)} f(e)$$

Flow must **respect capacity**.

$$0 \leq f(e) \leq c_e.$$



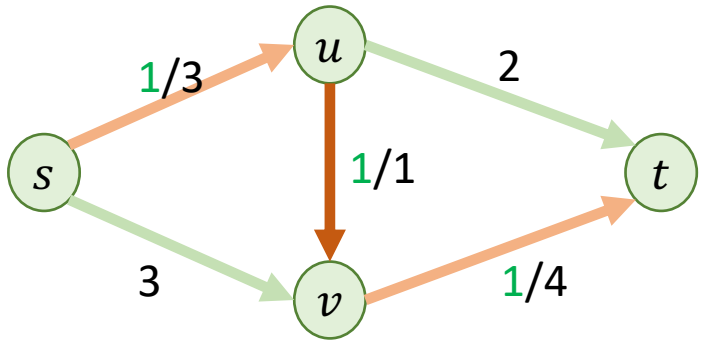
Max Flow, Constraints

Flow is **conserved** at each node other than s and t.

$$\sum_{e \in (\text{edges entering } u)} f(e) = \sum_{e \in (\text{edges leaving } u)} f(e)$$

Flow must **respect capacity**.

$$0 \leq f(e) \leq c_e.$$



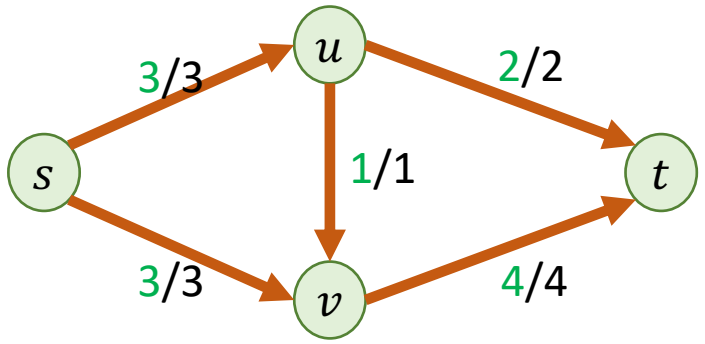
Max Flow, Constraints

Flow is **conserved** at each node other than s and t.

$$\sum_{e \in (\text{edges entering } u)} f(e) = \sum_{e \in (\text{edges leaving } u)} f(e)$$

Flow must **respect capacity**.

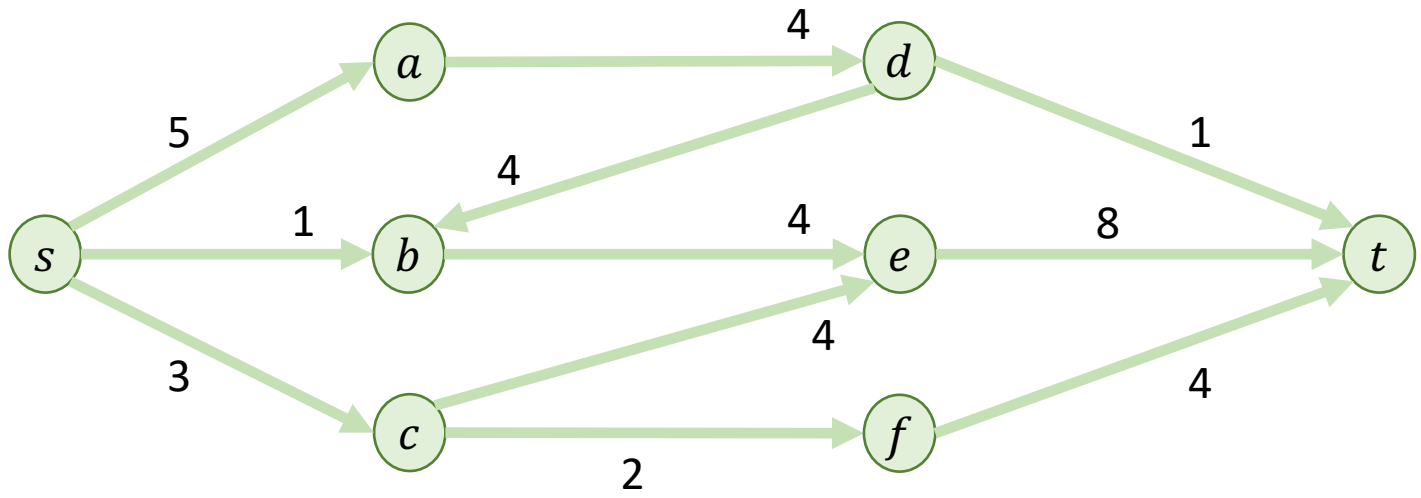
$$0 \leq f(e) \leq c_e.$$



In this example, the max flow will **saturate every edge**, but this is **not always possible** in every problem.

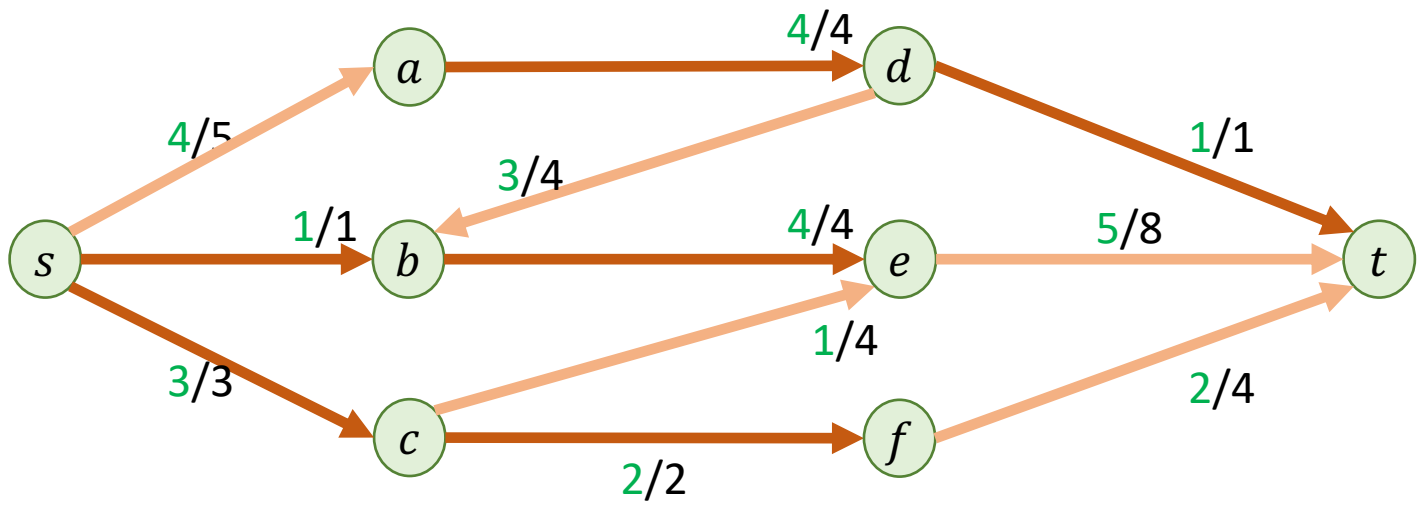
Max Flow Example

What is the **max flow** of the following network?



Max Flow Example

What is the **max flow** of the following network?

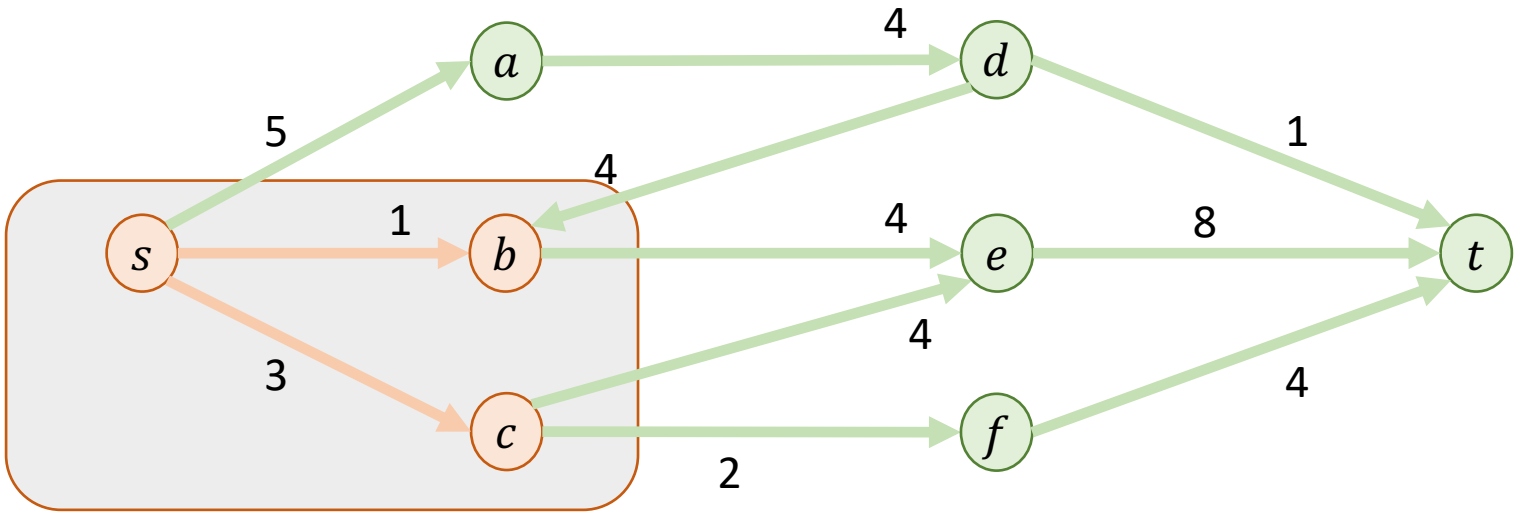


Example: Max flow value is 8.

Capacity of a Cut

A *cut* is a partition into **two sets of nodes**: s on one side; t on the other.

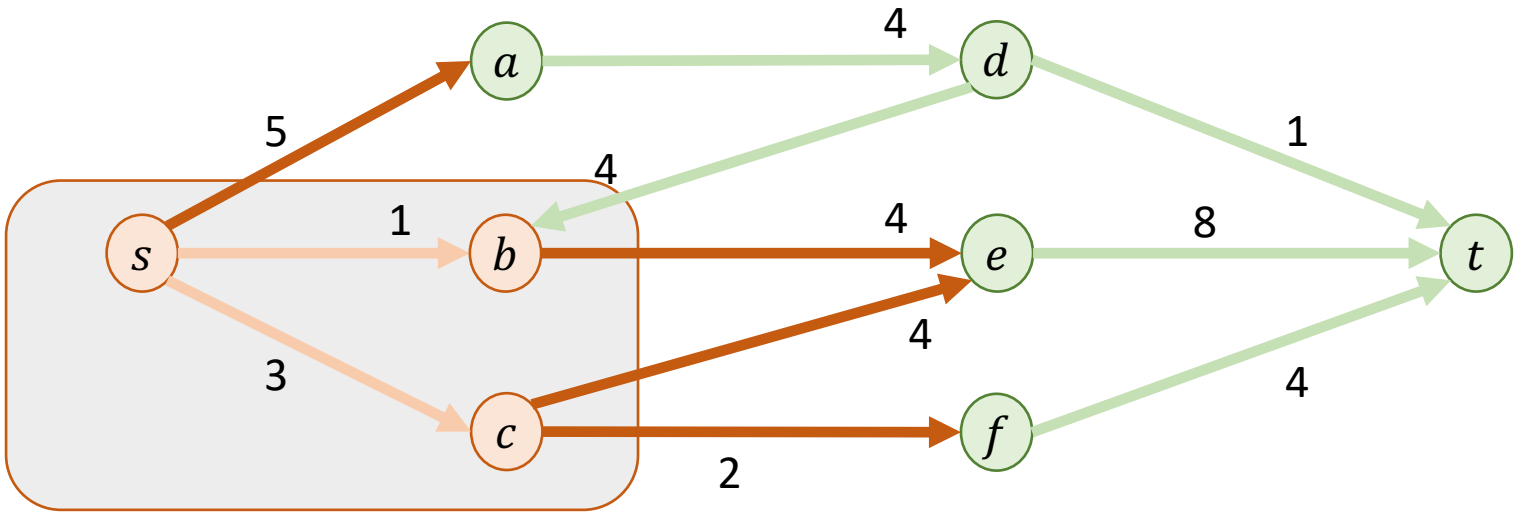
This is defined similarly to when we used cuts while computing **minimum spanning trees**.



The nodes inside a cut do not have to be touching.

Capacity of a Cut

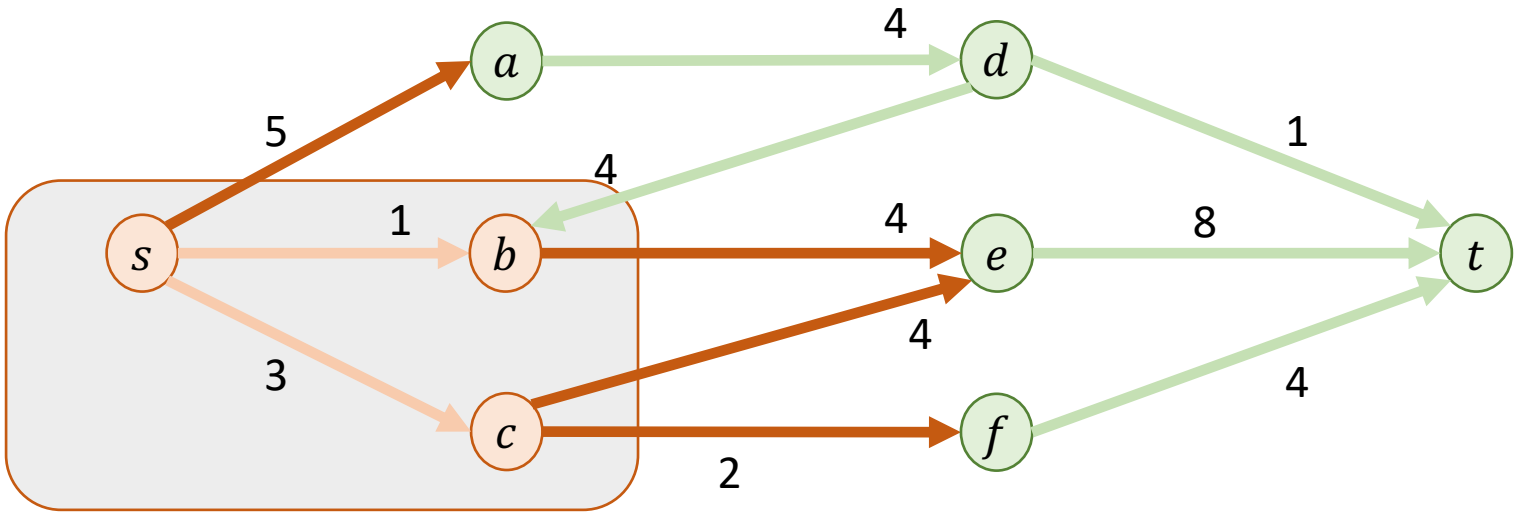
The *capacity of a cut* is the sum of the capacities **leaving a cut**.



Example: Capacity of the cut $= 5 + 4 + 4 + 2 = 15$.
Note that **edge** (d, b) **does not count** against the capacity.

Min Cut Problem

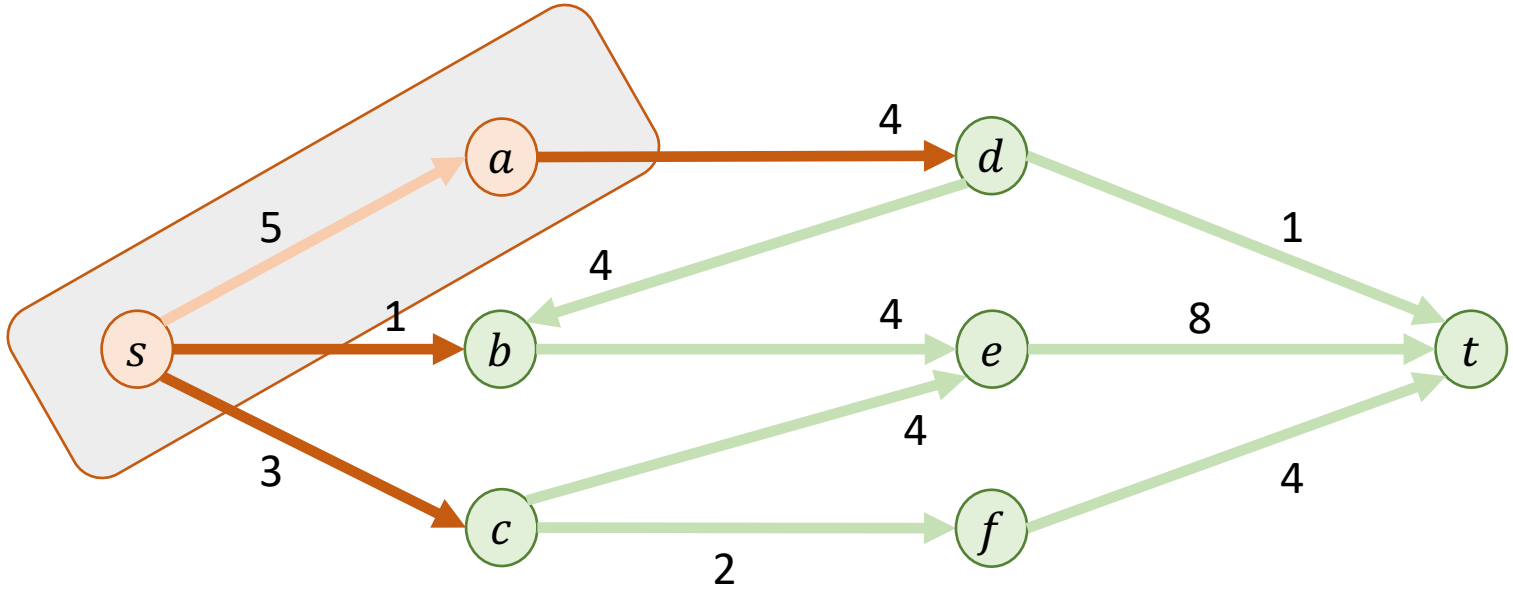
The *min cut problem* asks what is the **minimum capacity** over all cuts?



Example: What is the **bottleneck** in a data network?

Min Cut Problem

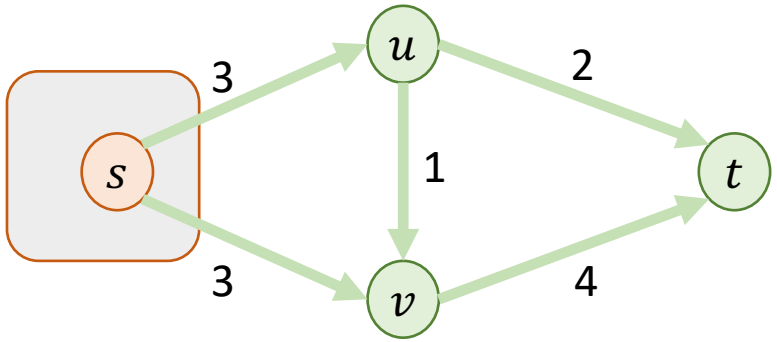
The *min cut problem* asks what is the **minimum capacity** over all cuts?



Example: Capacity $= 4 + 1 + 3 = 8$.

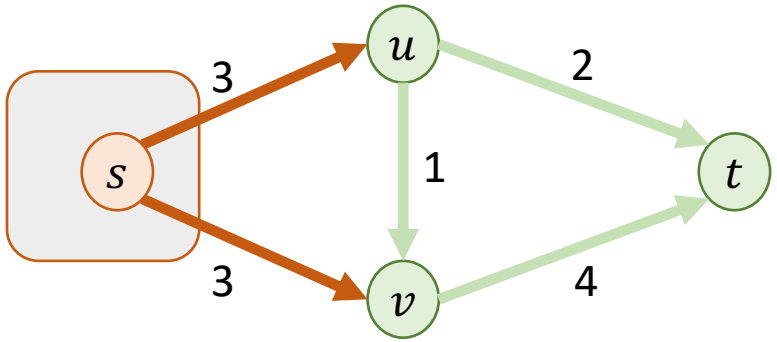
Min Cut Example

Example: What is the **capacity** of the cut shown?



Min Cut Example

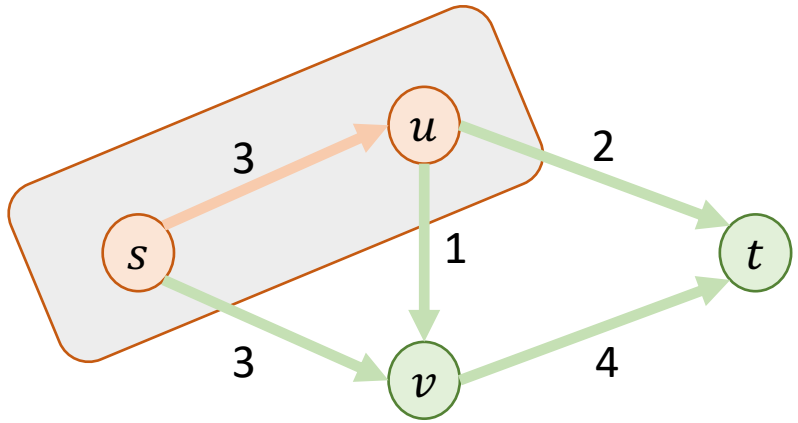
Example: What is the **capacity** of the cut shown?



Answer: $3 + 3 = 6$.

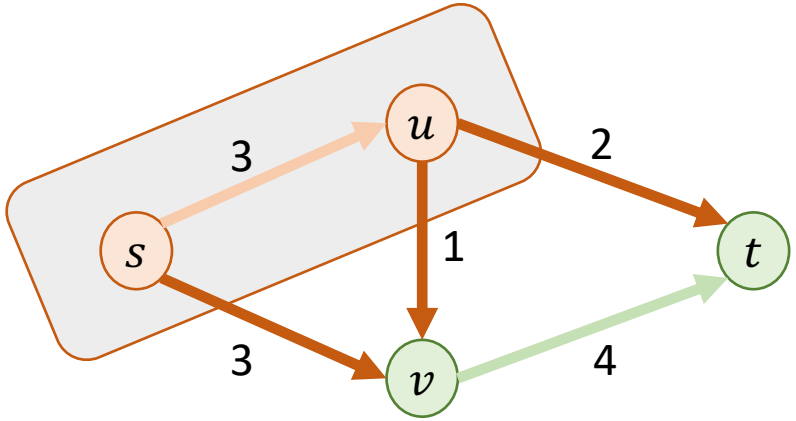
Min Cut Example

Example: What is the **capacity** of the cut shown?



Min Cut Example

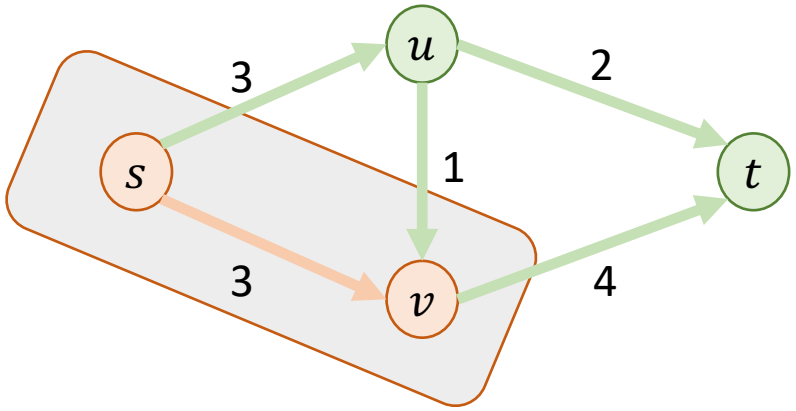
Example: What is the **capacity** of the cut shown?



Answer: $3 + 1 + 2 = 6$.

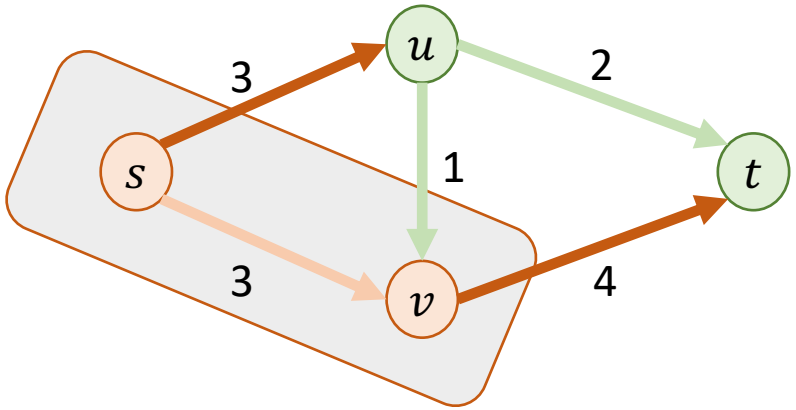
Min Cut Example

Example: What is the **capacity** of the cut shown?



Min Cut Example

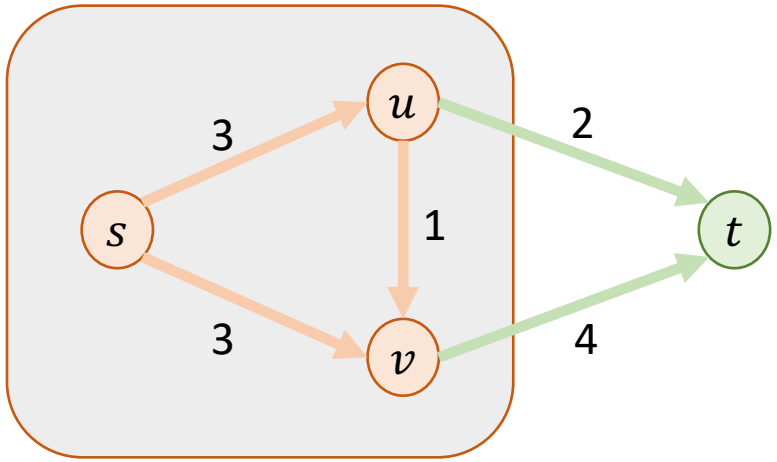
Example: What is the **capacity** of the cut shown?



Answer: $3 + 4 = 7$.

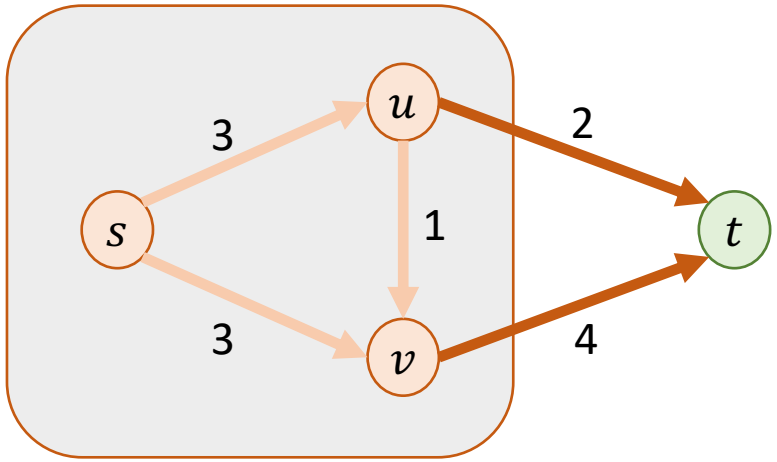
Min Cut Example

Example: What is the **capacity** of the cut shown?



Min Cut Example

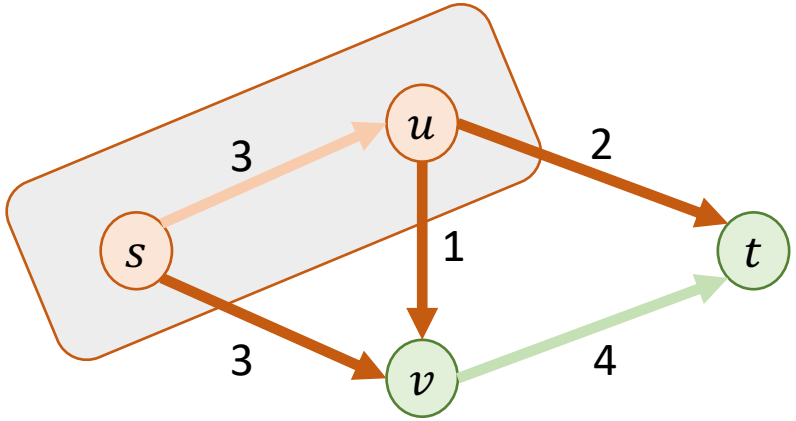
Example: What is the **capacity** of the cut shown?



Answer: $2 + 4 = 6$.

Min Cut Example

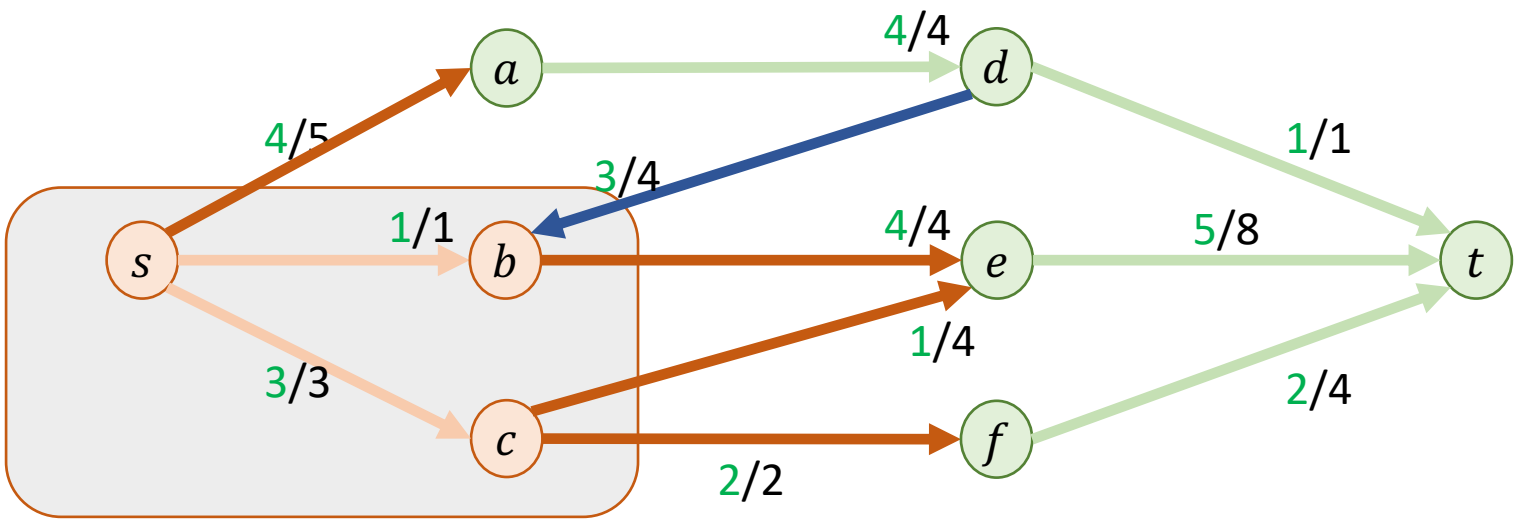
Example: The **min cut** for this network is 6, because of all possible cuts, these were the smallest.



Note that **multiple cuts** may have the **same** min cut capacity.

Net Flow Across a Cut

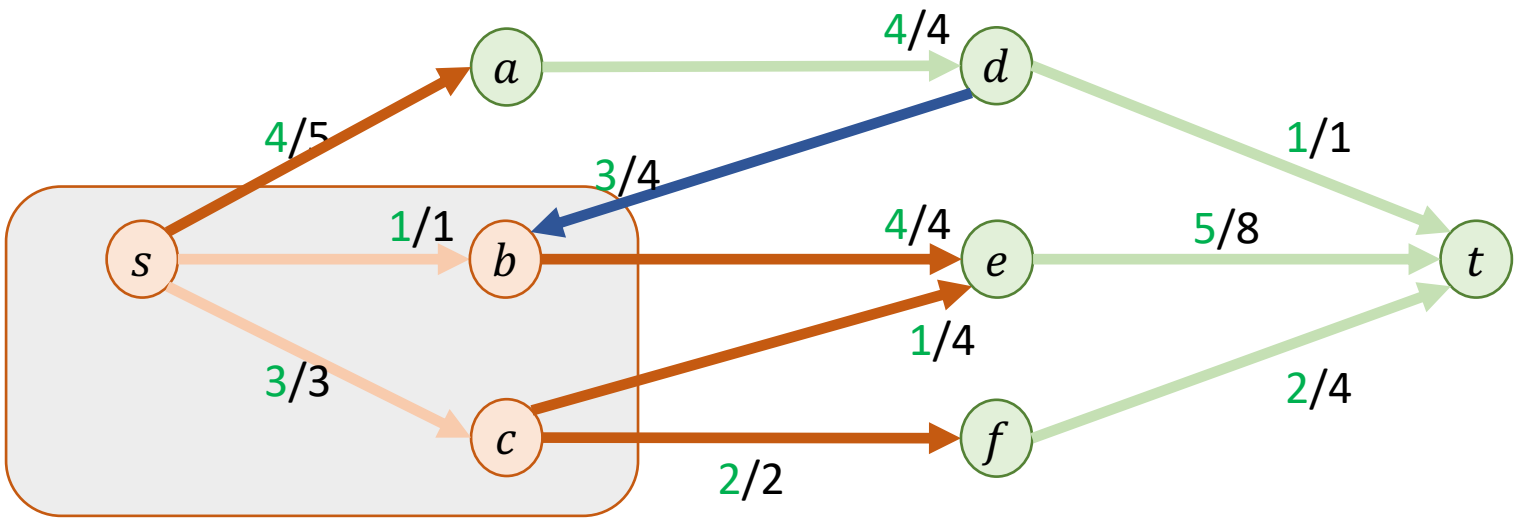
Net flow across a cut: difference between outgoing and incoming flows.



Net flow $= \sum_{e \in (edges\ leaving\ cut)} f(e) - \sum_{e \in (edges\ entering\ cut)} f(e)$

Net Flow Across a Cut

Net flow across a cut: difference between outgoing and incoming flows.

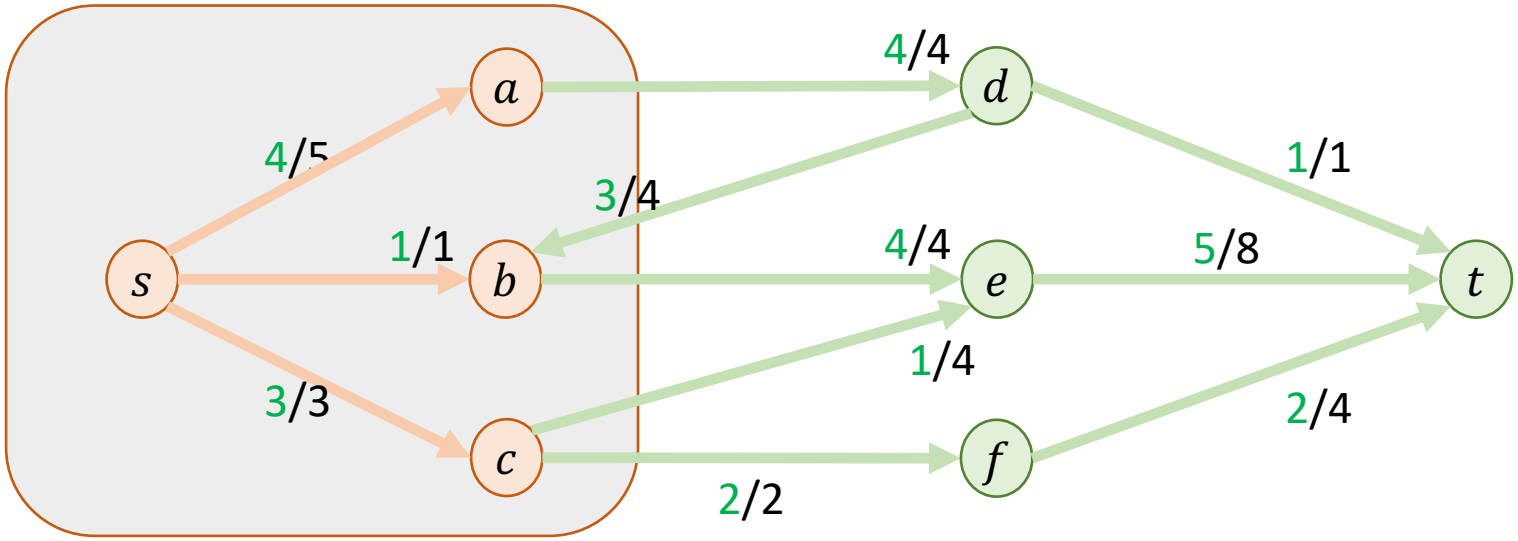


Net flow $= \sum_{e \in (edges\ leaving\ cut)} f(e) - \sum_{e \in (edges\ entering\ cut)} f(e)$

Example: Net flow $= (4 + 4 + 1 + 2) - (3) = 8.$
Capacity $= 5 + 4 + 4 + 2 = 15.$

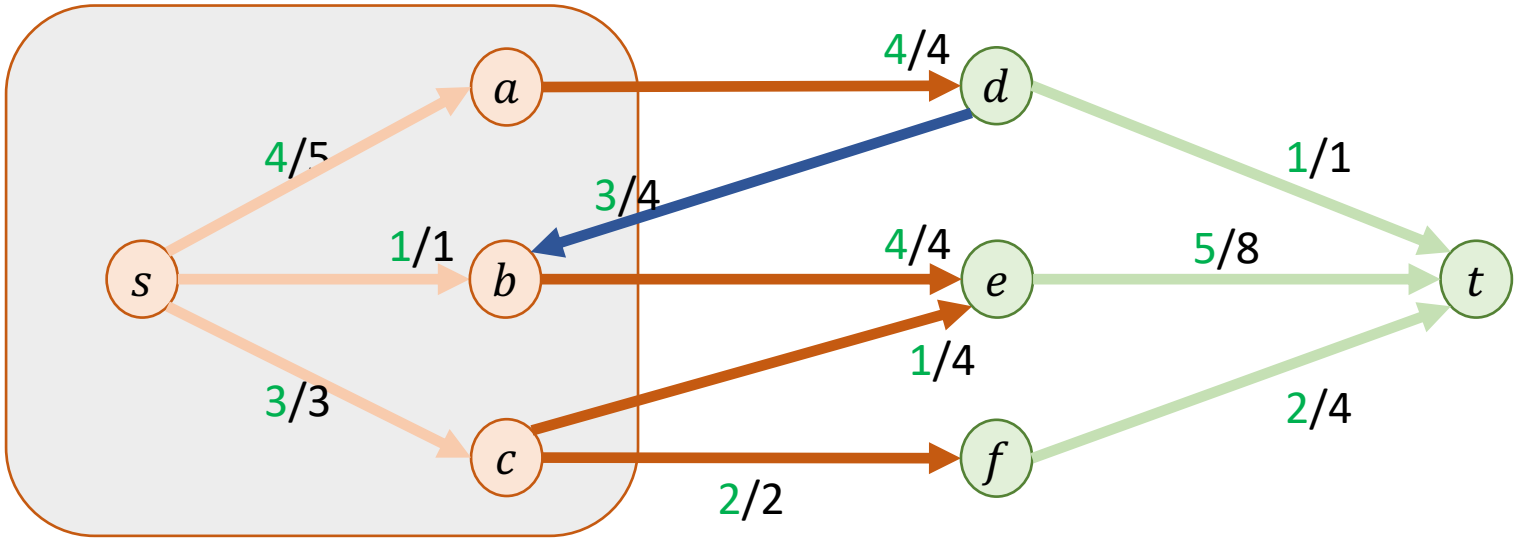
Net Flow Example

What is the **net flow** across the following cut?



Net Flow Example

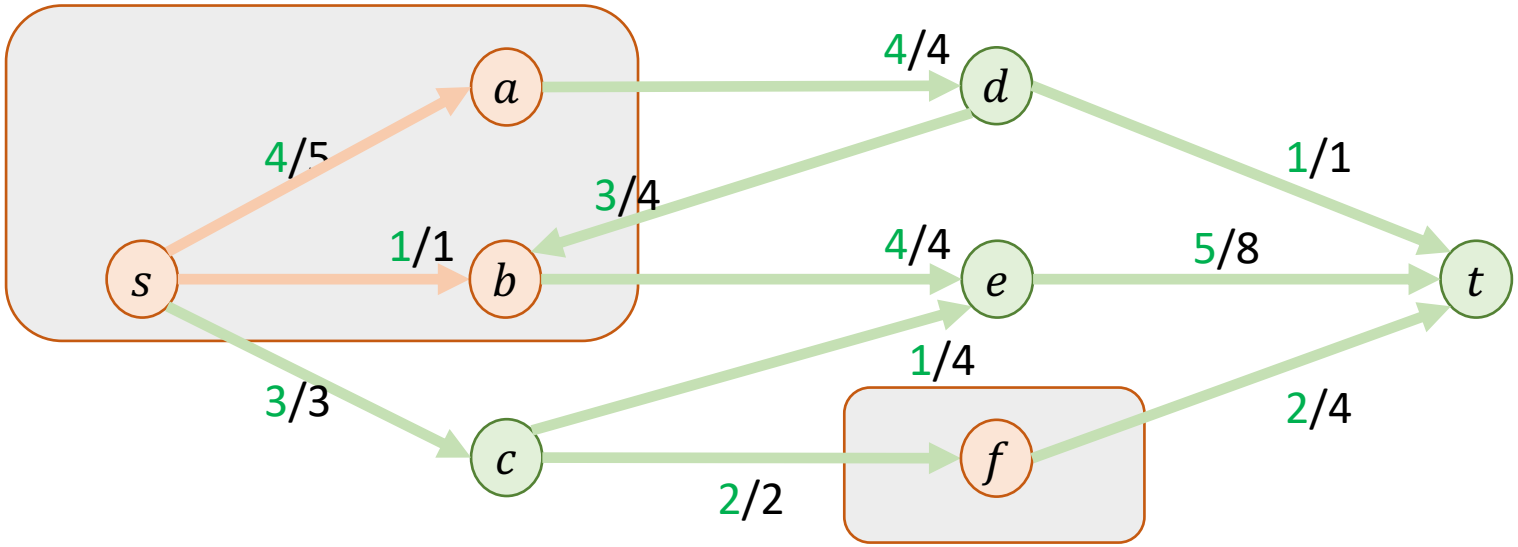
What is the **net flow** across the following cut?



Answer: Net Flow $= (4 + 4 + 1 + 2) - (3) = 8$.
Capacity $= 4 + 4 + 4 + 2 = 14$.

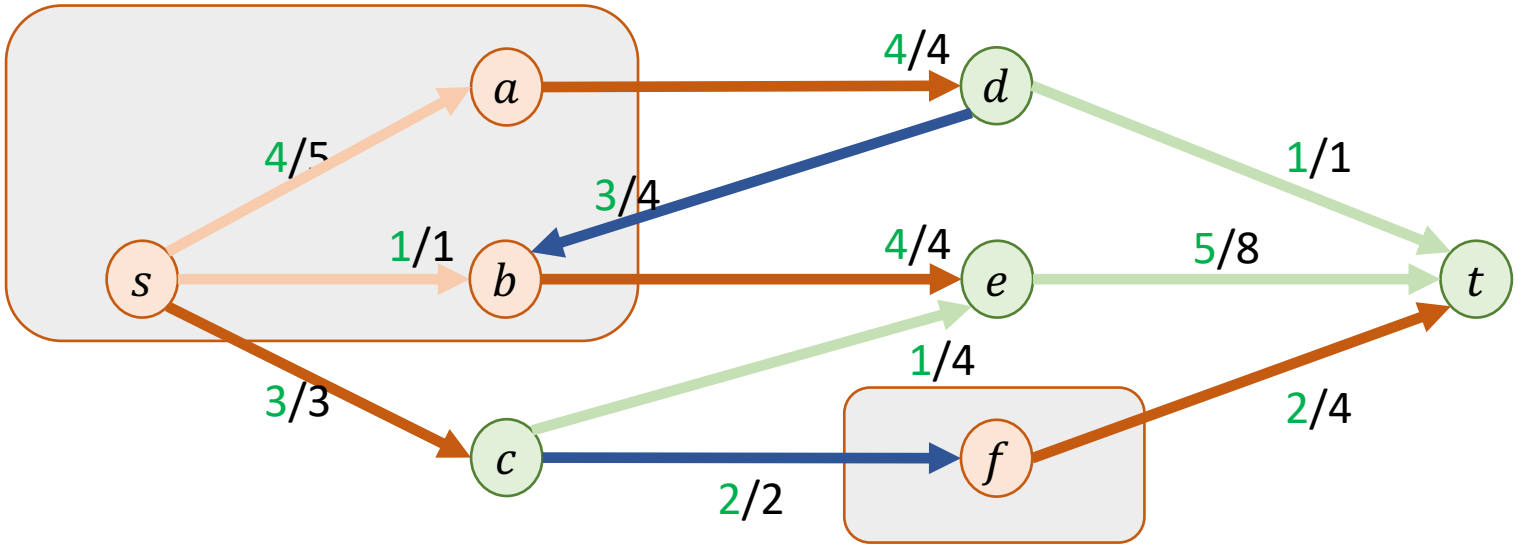
Net Flow Example

What is the **net flow** across the following cut?



Net Flow Example

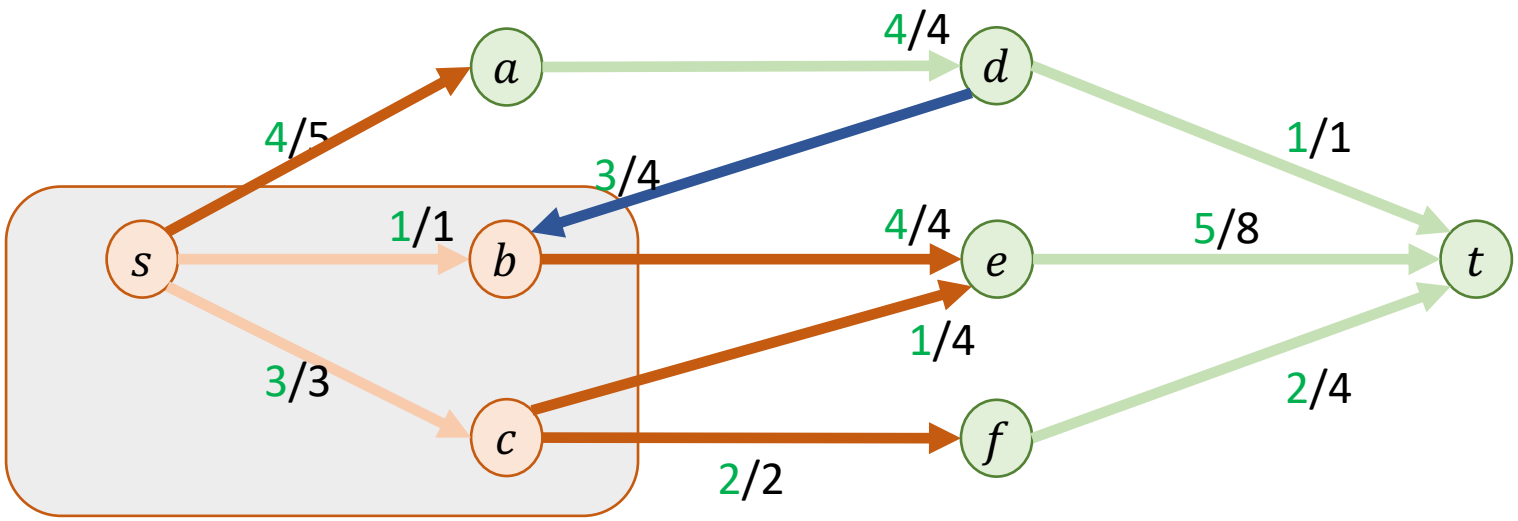
What is the **net flow** across the following cut?



Answer: Net flow $= (4 + 4 + 3 + 2) - (3 + 2) = 8.$
Capacity $= 4 + 4 + 3 + 4 = 15.$

Net Flow Across a Cut

Net flow across a cut: difference between outgoing and incoming flows.

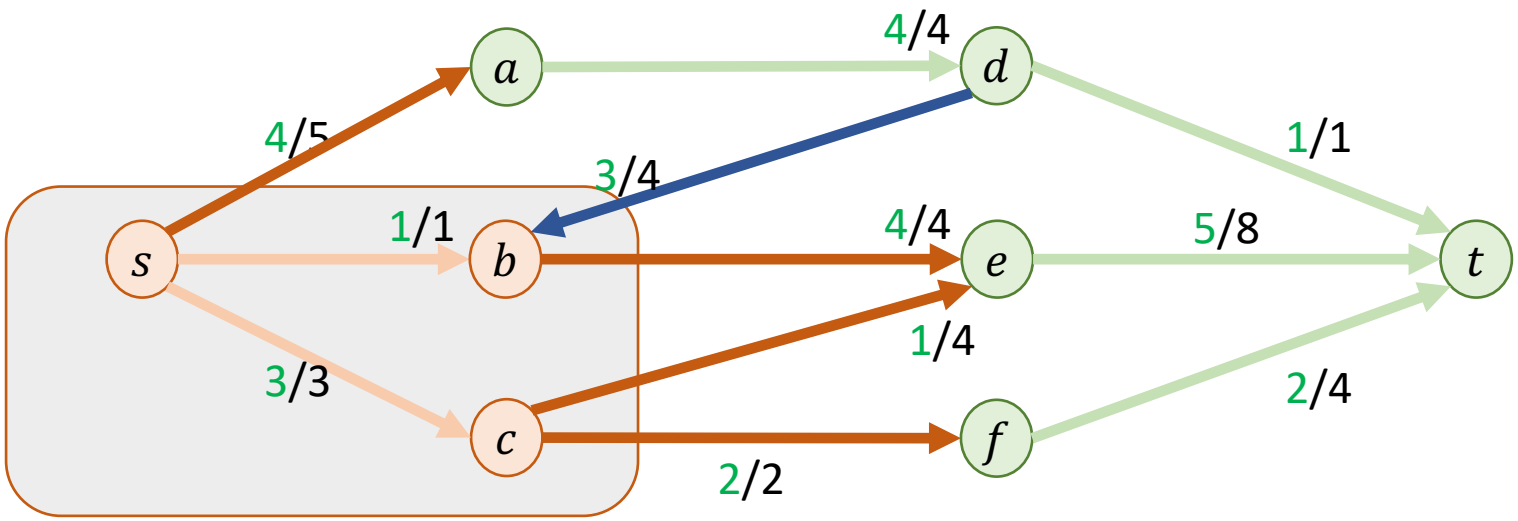


Observation 1: The **net flow will always be the same** no matter which cut we choose.

- This can be deduced from the **conservation** principle as we add or remove nodes.

Net Flow Across a Cut

Net flow across a cut: difference between outgoing and incoming flows.

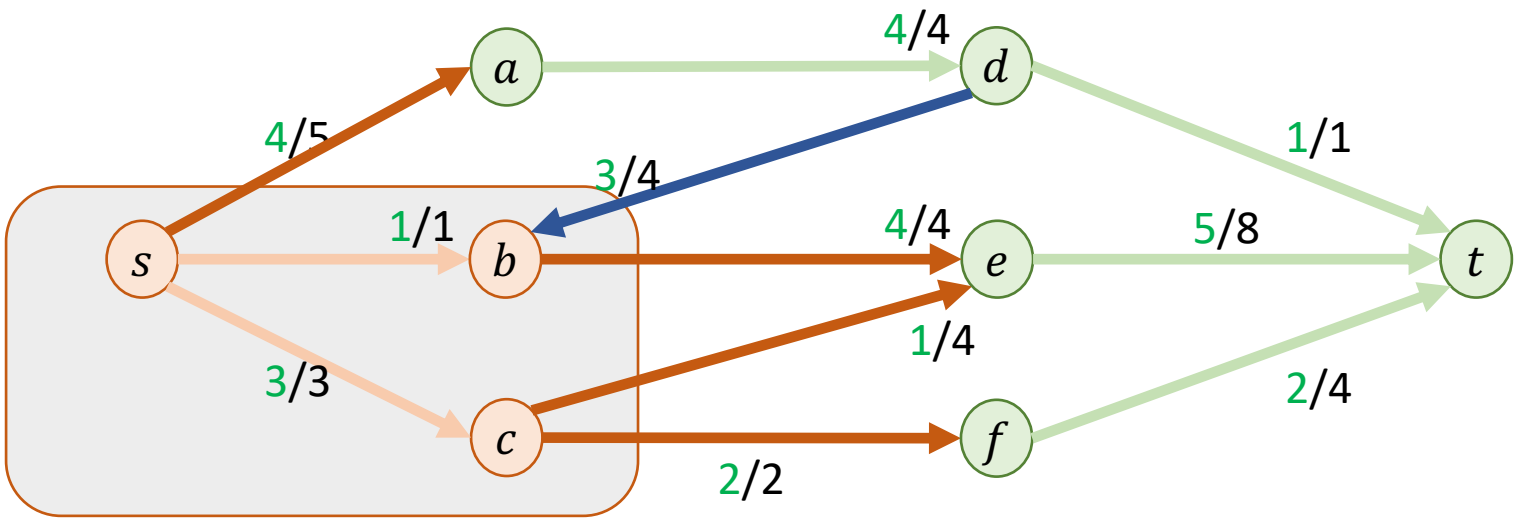


Observation 2: The **max flow cannot be larger** than the capacity of a cut.

- The capacity is an **upper bound** for net flow.

Net Flow Across a Cut

Net flow across a cut: difference between outgoing and incoming flows.



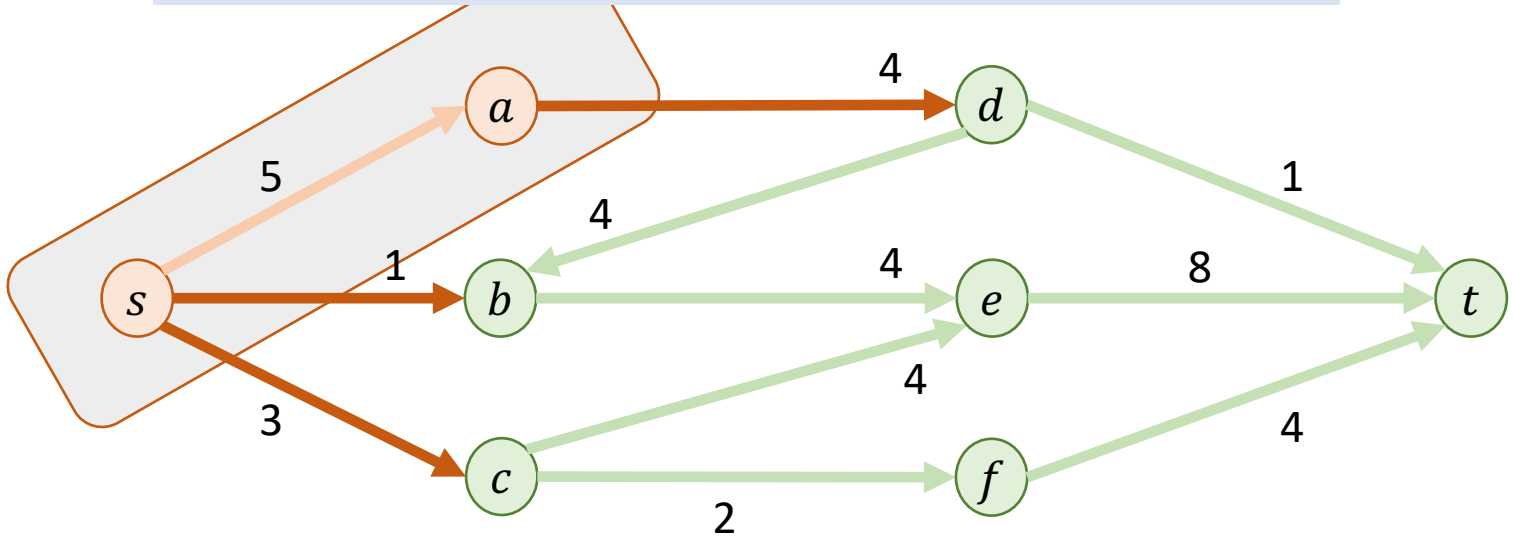
Observation 3: Some cut must have a capacity which **equals the max flow**.

- If not, then we could have still **sent more flow** across the network.

Max Flow/Min Cut Duality

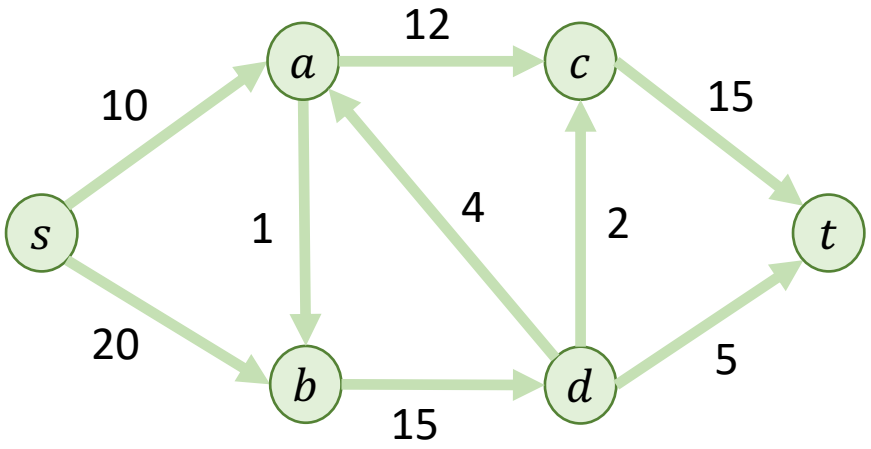
Conclusion: the **max flow** value equals the **min cut** value!

If we can compute one, then we have computed the other.



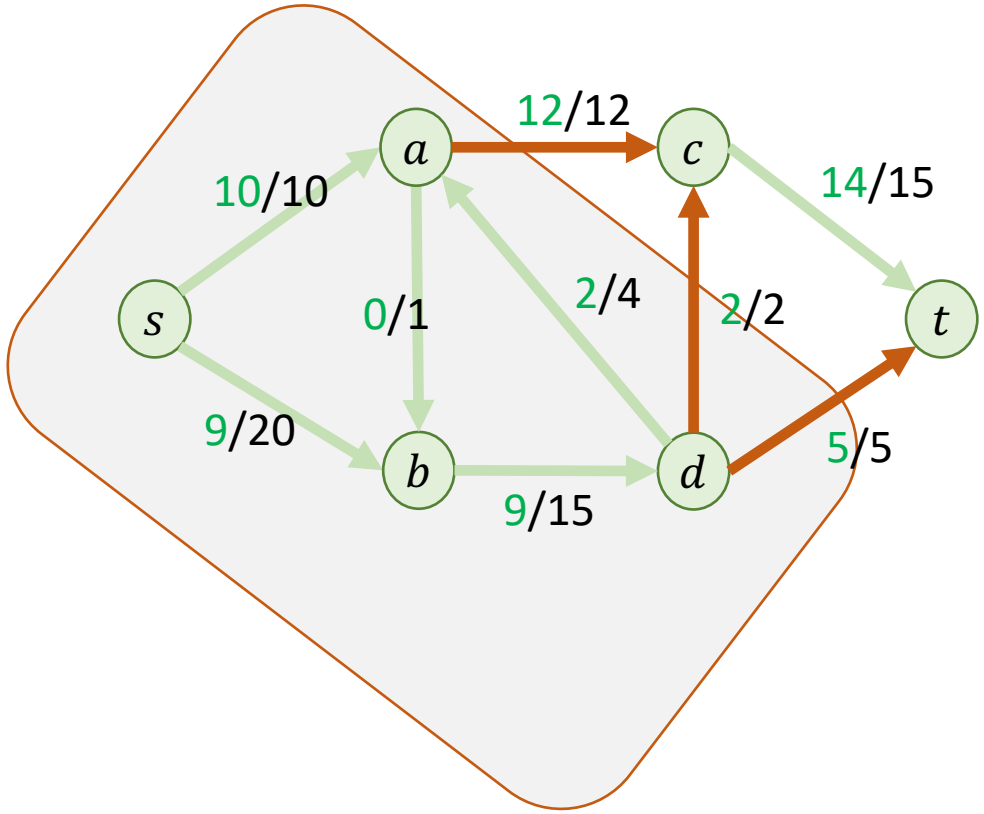
Practice

Find a **max flow** and a **min cut** for the following graph:



Practice

Find a **max flow** and a **min cut** for the following graph:



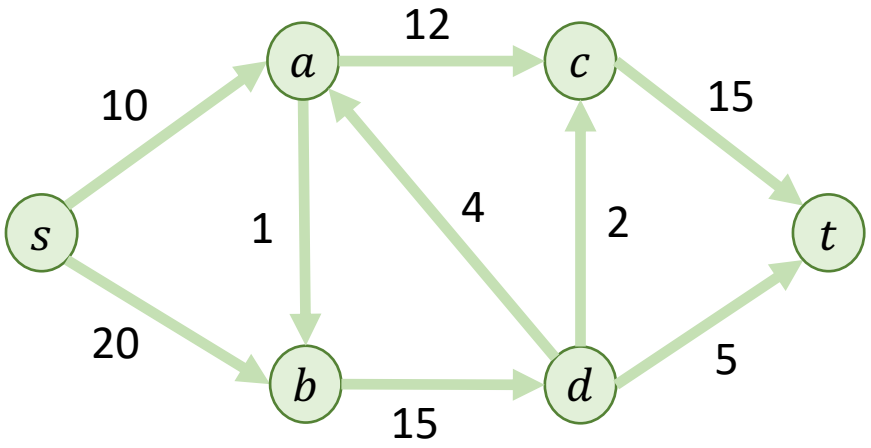
Flow value = 19

Min cut capacity = 19

Algorithmic Max Flow: Baby Steps

If we want to find max flow, we would prefer to use an **algorithm**.

Before we produce an algorithm for **max flow**, let's start simple… can we find **any** flow through this graph?



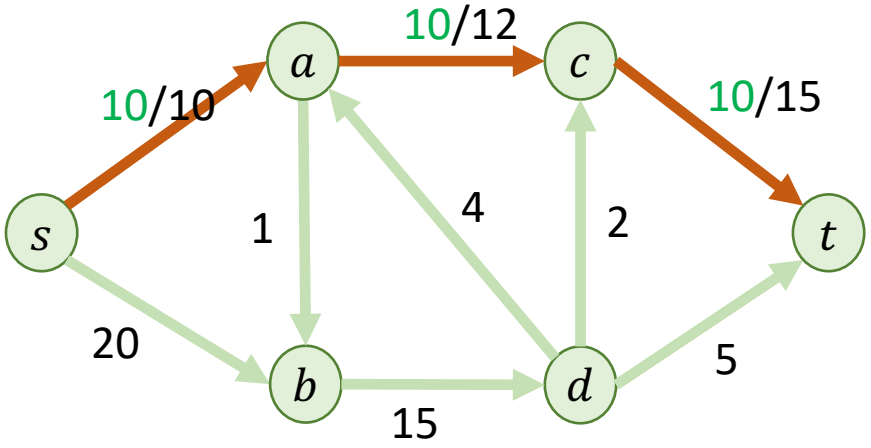
Algorithmic Max Flow: Baby Steps

If we want to find max flow, we would prefer to use an **algorithm**.

Before we produce an algorithm for **max flow**, let's start simple… can we find **any** flow through this graph?

Suggestion: Let us find **any path** from source to sink. That path implies a **flow**.

How would we **find a path** from source to sink?

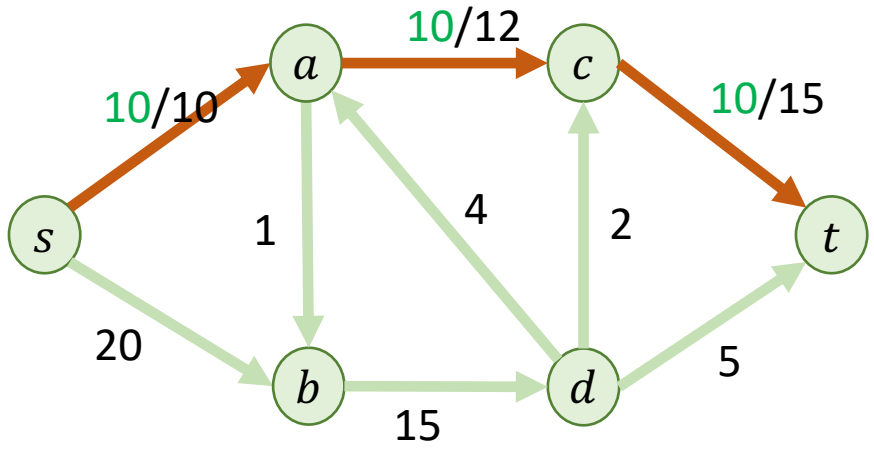


Algorithmic Max Flow: Baby Steps

If we want to find max flow, we would prefer to use an **algorithm**.

Before we produce an algorithm for **max flow**, let's start simple… can we find **any** flow through this graph?

Suggestion: Let us find **any path** from source to sink. That path implies a **flow**.



How would we **find a path** from source to sink?

Our ordinary **graph search algorithms** will find one.

- **Breadth first search** (BFS).
- **Depth first search** (DFS).

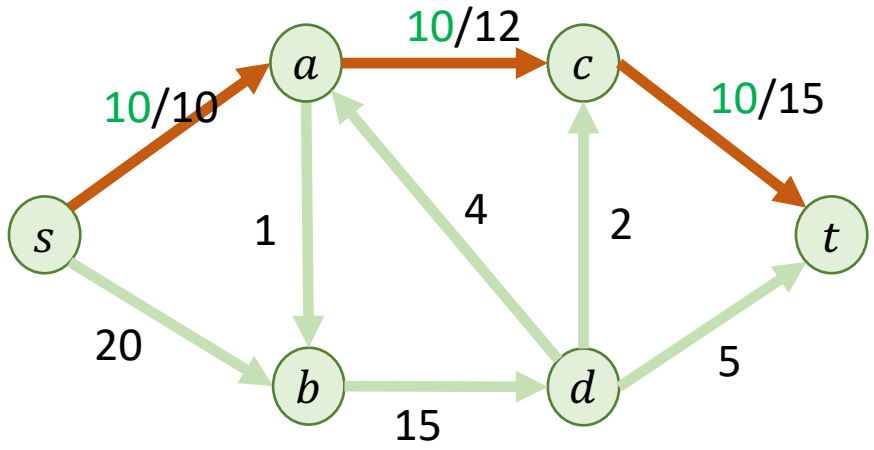
On the other hand, **if there is no path**, then there **is no possible flow**.

Algorithmic Max Flow: Baby Steps

If we want to find max flow, we would prefer to use an **algorithm**.

Before we produce an algorithm for **max flow**, let's start simple… can we find **any** flow through this graph?

Suggestion: Let us find **any path** from source to sink. That path implies a **flow**.



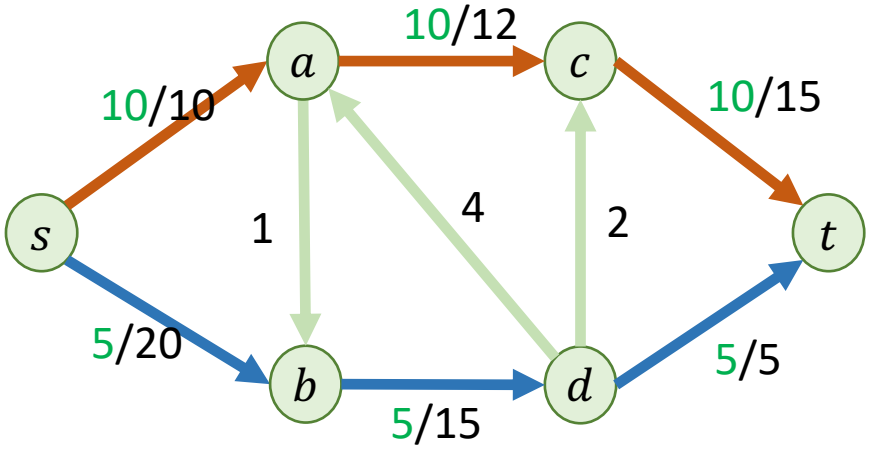
We will **prefer a BFS**, for reasons we shall see later.

If we find a path, we can use a **bottleneck edge** to **maximize flow along that path**.

Algorithmic Max Flow: Augmenting

If we want to find max flow, we would prefer to use an **algorithm**.

Observation: If we find a **second path**, we can combine it with the first to form a **larger flow**, provided that the two together do not violate any capacities.



Algorithmic Max Flow: Augmenting

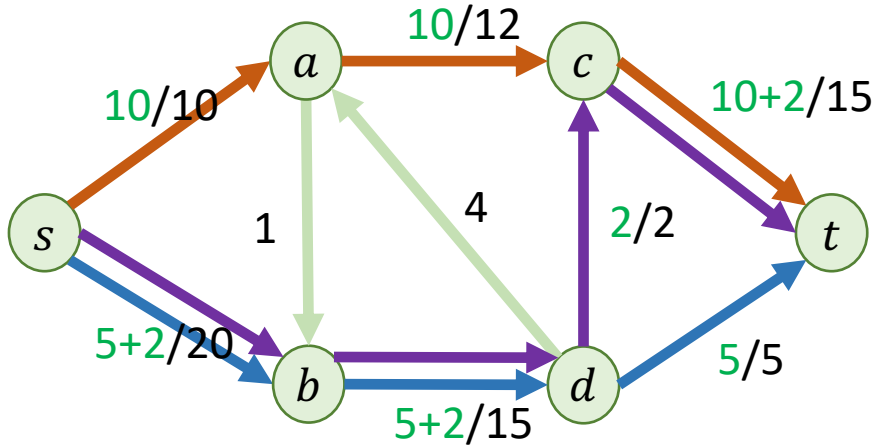
If we want to find max flow, we would prefer to use an **algorithm**.

Observation: If we find a **second path**, we can combine it with the first to form a **larger flow**, provided that the two together do not violate any capacities.

This works even if the new path **overlaps an existing path**.

These added paths are called *augmenting paths*.

- Augmenting paths will **only increase** flow value.



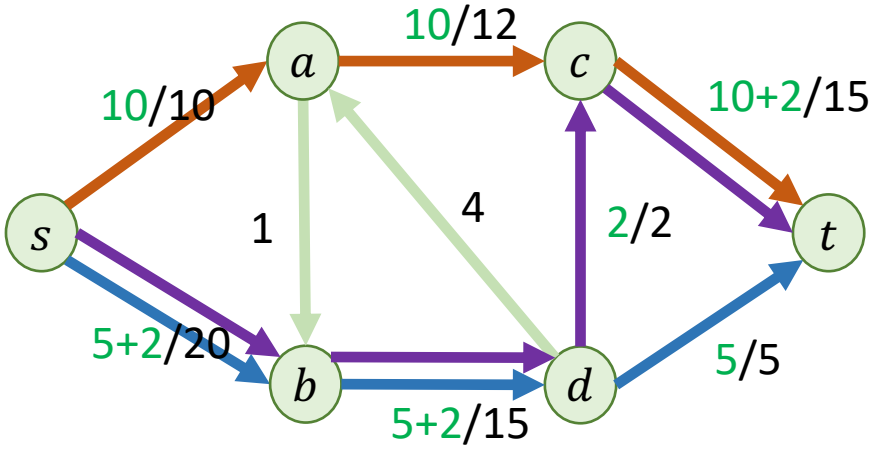
Algorithmic Max Flow: Augmenting

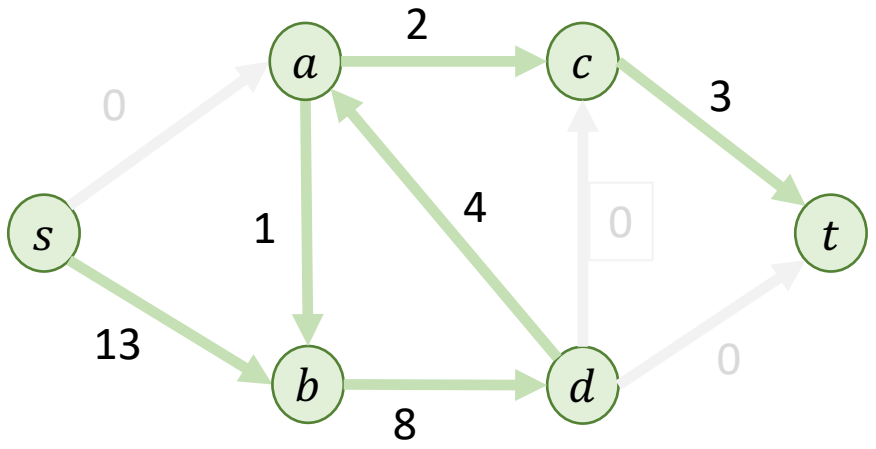
If we want to find max flow, we would prefer to use an **algorithm**.

How would we **find new paths** once we have already used some of the paths?

- We do not want to **waste time revisiting paths** which are already used.

Suggestion: Consider only the capacity which is "**left over**" after using known paths.





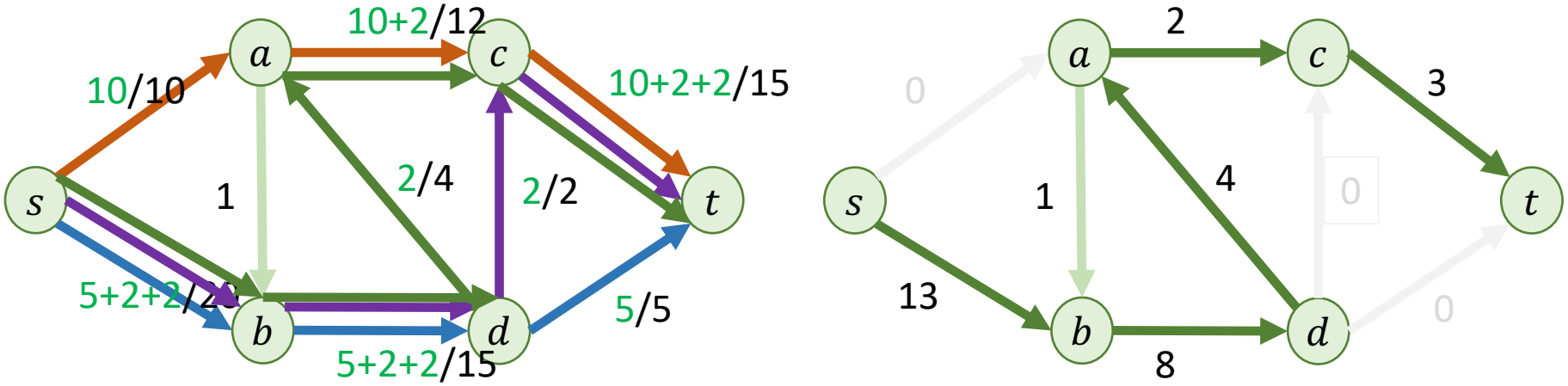
Algorithmic Max Flow: Augmenting

If we want to find max flow, we would prefer to use an **algorithm**.

How would we **find new paths** once we have already used some of the paths?

- We do not want to **waste time revisiting paths** which are already used.

Suggestion: Consider only the capacity which is "**left over**" after using known paths.



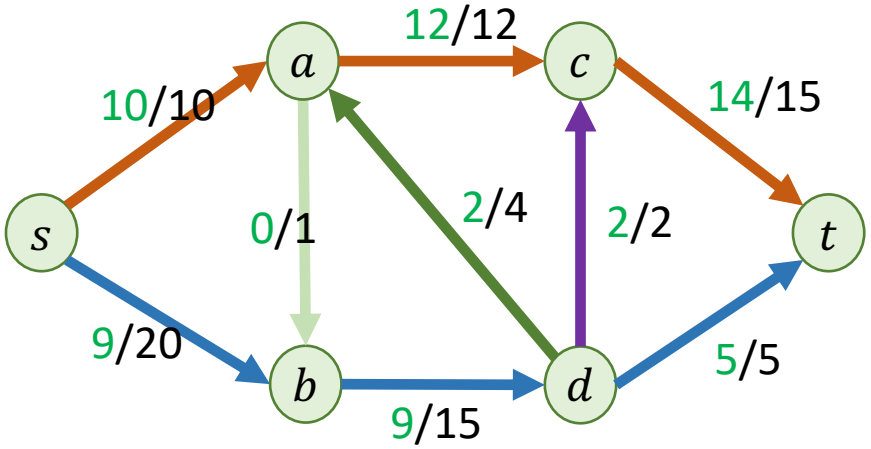
Algorithmic Max Flow: Augmenting

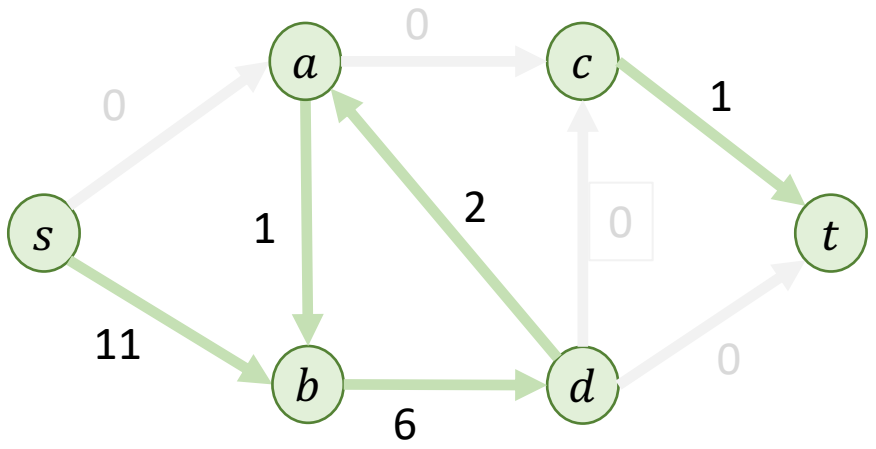
If we want to find max flow, we would prefer to use an **algorithm**.

How would we **find new paths** once we have already used some of the paths?

- We do not want to **waste time revisiting paths** which are already used.

Suggestion: Consider only the capacity which is "**left over**" after using known paths.



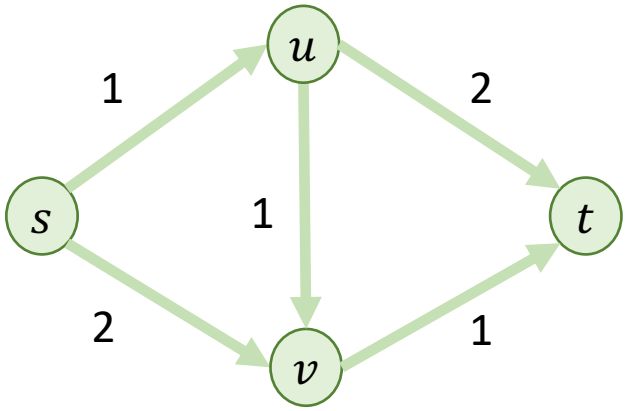


Note that there is now **no more path** from source to sink.

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:

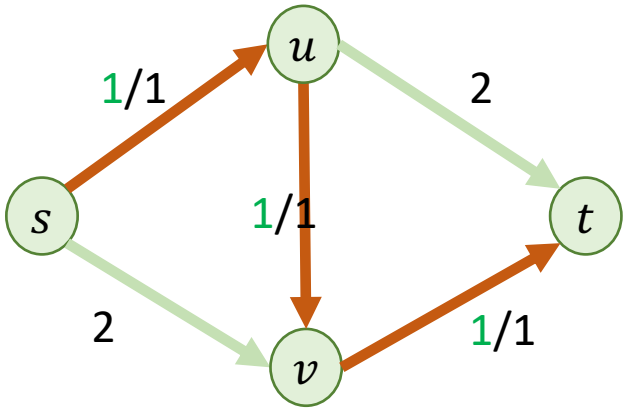


What is the max flow here?

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:

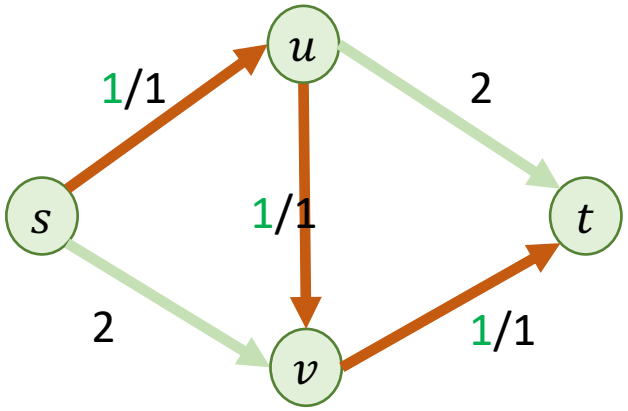


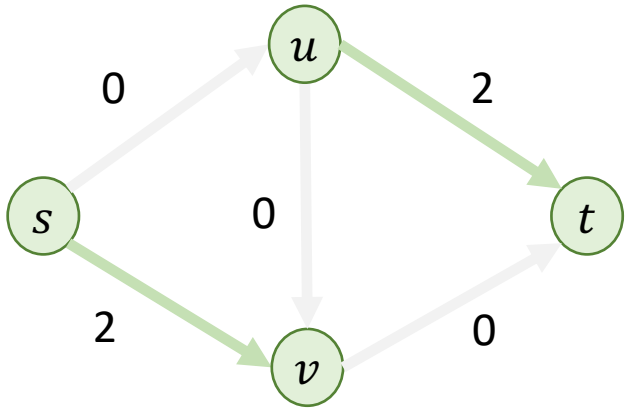
What is the max flow here?

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:





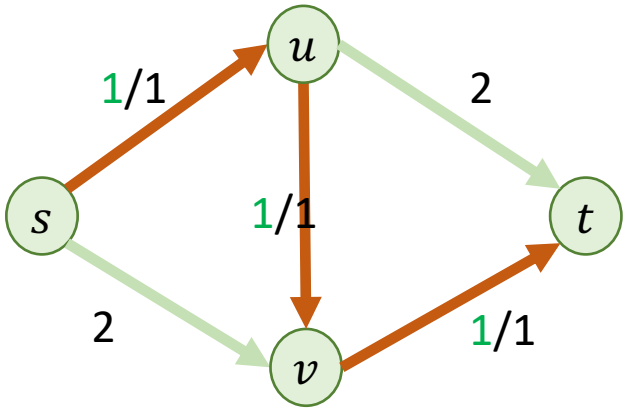
What is the max flow here?

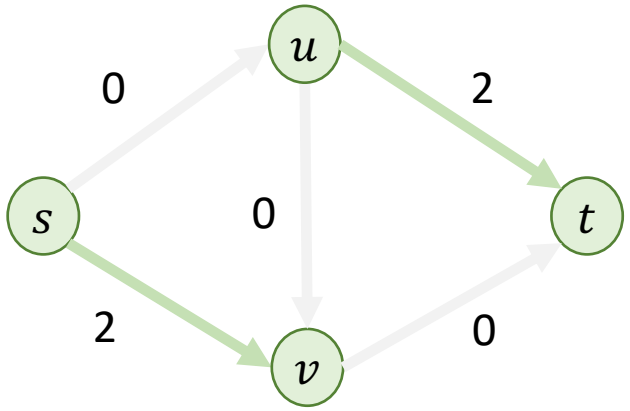
There are **no more paths to find**, yet we can easily see that the true **max flow is better than this**!

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:





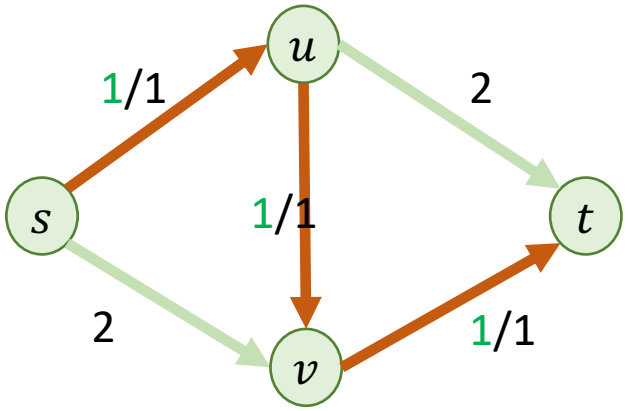
What is the max flow here?

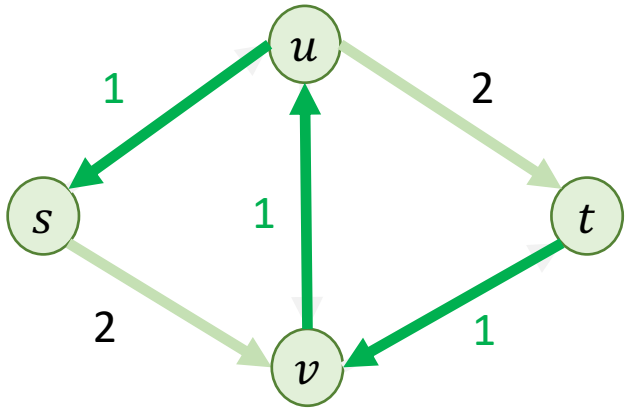
The best result does not use the middle (u, v) edge.
Is there a way for us to **undo** a selection?

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:





What is the max flow here?

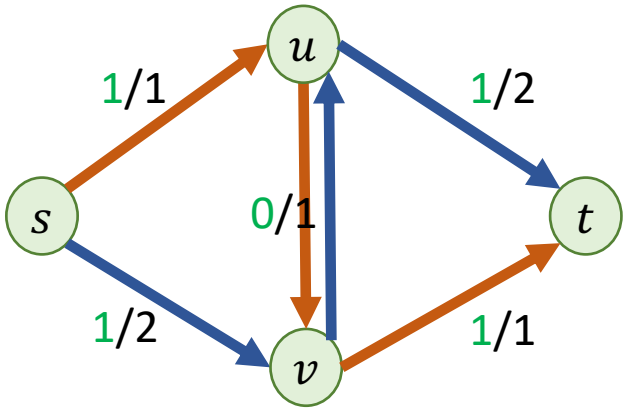
Solution: We can create a virtual edge in the **opposite direction** to show **how much less flow** we could have had across that edge. This lets us **undo** a selection.

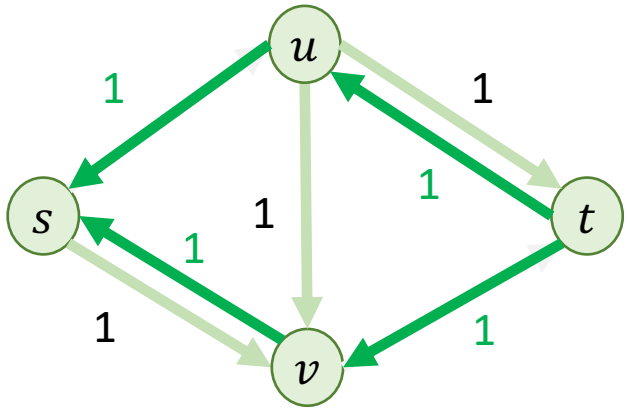
It is **still true** that **any path** from source to sink **increases the flow** value.

Reliably Finding Augmenting Paths

If we always **removed used capacity** from the graph whenever we find a flow, will that always **lead us to a max flow**?

Maybe that is not enough. **Consider**:





Now, our "left over" graph – called a *residual graph* – has **no remaining *s*-*t* path**.

- We can truly say that **no more flow** can be added – we have a max flow!

Ford-Fulkerson Method

The *Ford-Fulkerson Method* for **finding a max flow** is as follows:

Repeat:

- Find an **augmenting path** (e.g. using a search algorithm from s to t).
- **If found**:
  - **Add the augmenting path** to the flow.
  - Recompute a **residual graph**.

…until no more augmenting paths can be found.

Min Cut Algorithm

Now that we know max flow, can we use it to find **min cut**?

Hint: What is the **relationship between** max flow and min cut?

Min Cut Algorithm

Now that we know max flow, can we use it to find **min cut**?

Hint: What is the **relationship between** max flow and min cut?

Max flow and min cut must have the **same value**.

Hint: What is true of the **capacity** at a min cut?

Min Cut Algorithm

Now that we know max flow, can we use it to find **min cut**?

Hint: What is the **relationship between** max flow and min cut?

Max flow and min cut must have the **same value**.

Hint: What is true of the **capacity** at a min cut?

The capacity at the min cut **equals the max flow** value.

- In fact, the **net flow** must be the same as well.

Hint: What is true of the **residual graph** relative to the min cut?

Min Cut Algorithm

Now that we know max flow, can we use it to find **min cut**?

Hint: What is the **relationship between** max flow and min cut?

Max flow and min cut must have the **same value**.

Hint: What is true of the **capacity** at a min cut?

The capacity at the min cut **equals the max flow** value.

- In fact, the **net flow** must be the same as well.

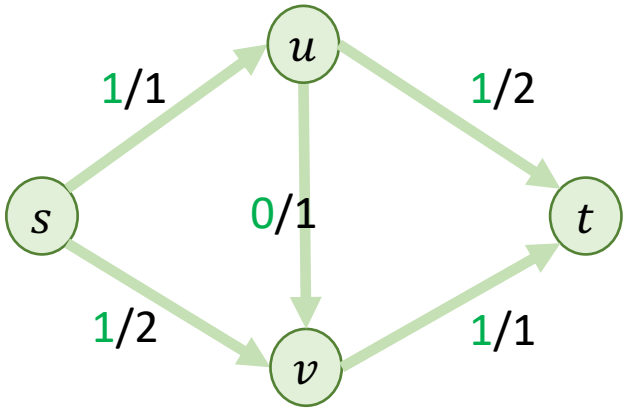
Hint: What is true of the **residual graph** relative to the min cut?

The flow must have **saturated the cut**, so the residual graph will have **no edges leaving** the cut.

- **Conclusion**: We can find a min cut by **finding max flow** and **computing the residual graph**.
- **The set of nodes which are still reachable will form a min cut.**

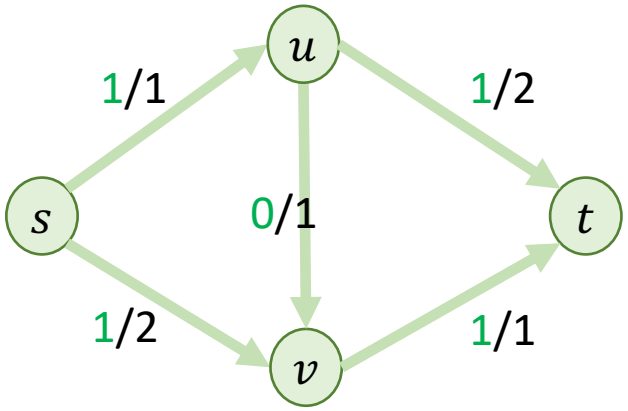
Min Cut Algorithm

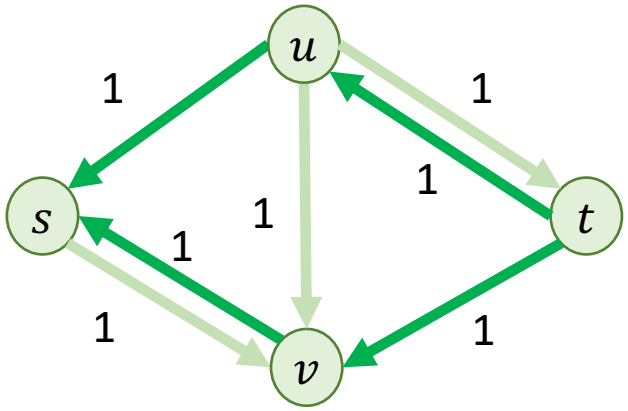
1. Find max flow.
2.
3.



Min Cut Algorithm

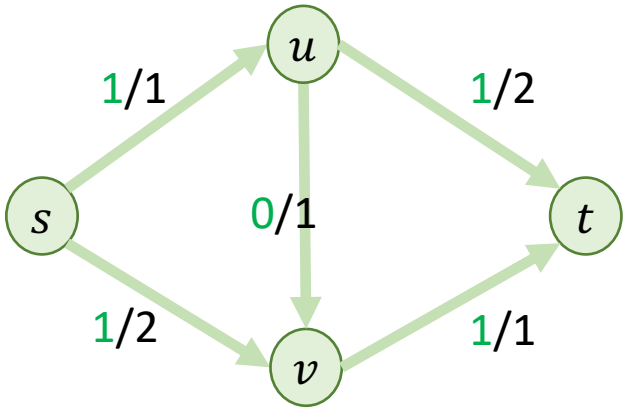
1. Find max flow.
2. Compute residual graph.
3.

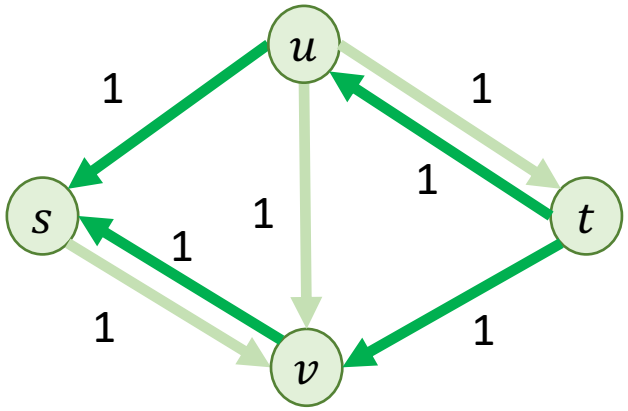


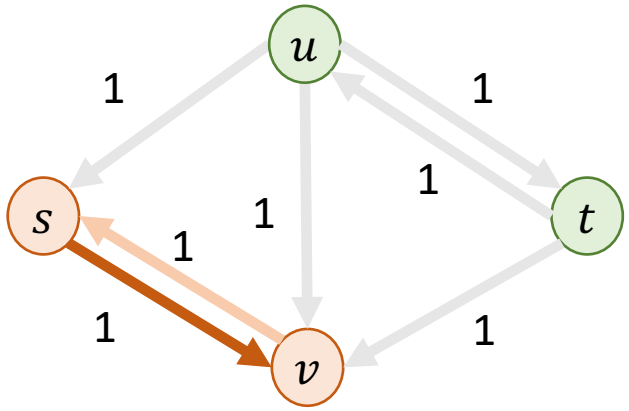


Min Cut Algorithm

1. Find max flow.
2. Compute residual graph.
3. Perform a BFS or DFS to find a min cut.

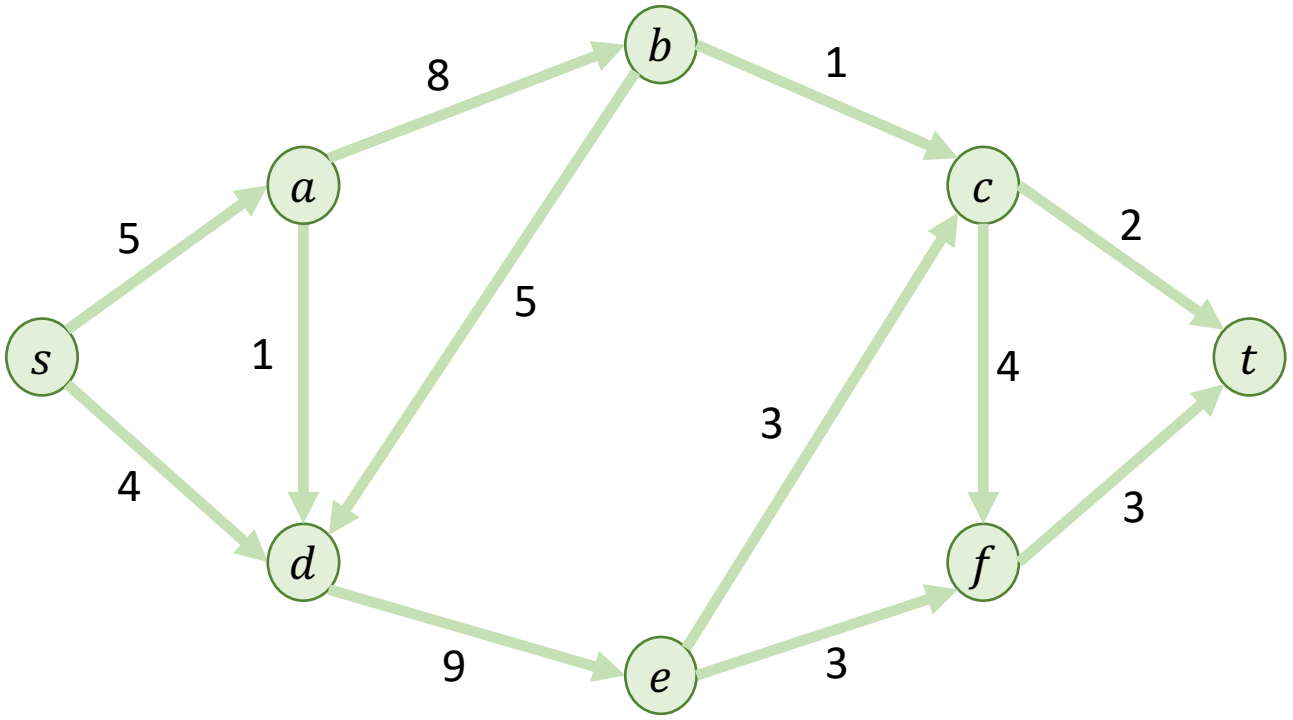






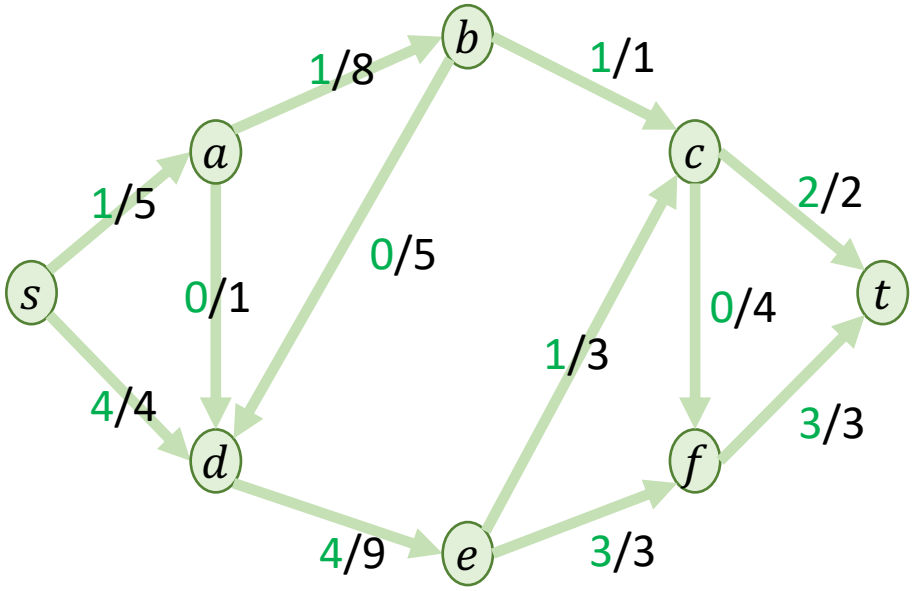
Practice

Find a **max flow**, **residual graph**, and **min cut** for the following graph:

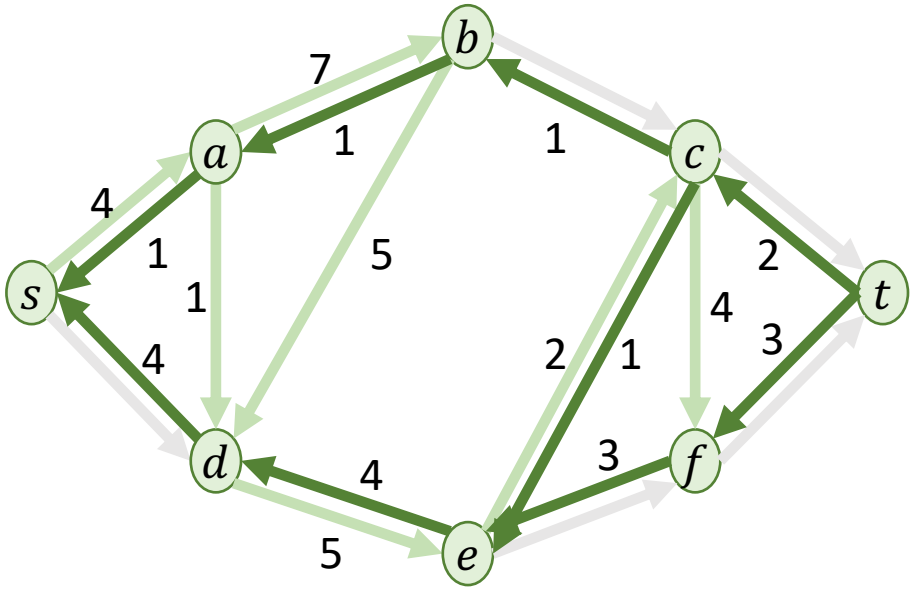


Practice

Find a **max flow**, **residual graph**, and **min cut** for the following graph:



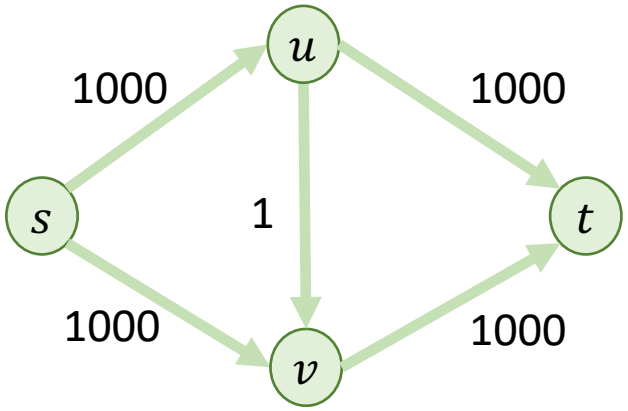
Max flow graph
Max flow value = 5.



Residual graph
Min cut is $\{s, a, b, c, d, e, f\}$ vs $\{t\}$.
Min cut value = 5.

Efficiently Finding Augmenting Paths

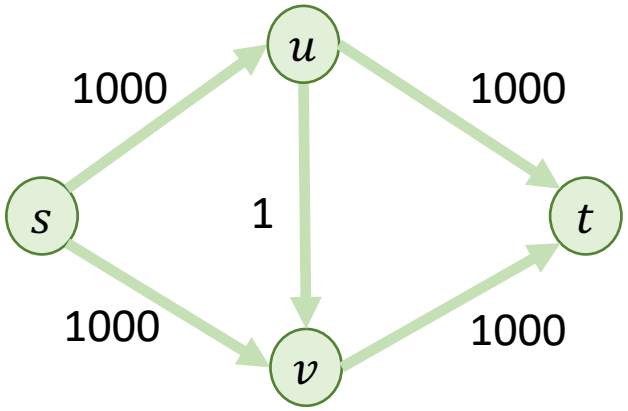
How long will it take Ford-Fulkerson to find max flow? That may depend partly on luck…

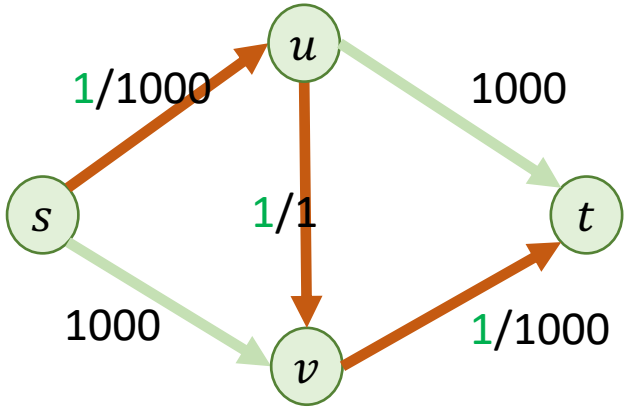


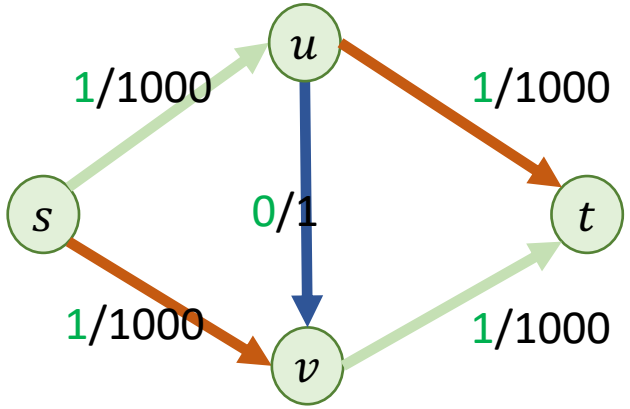
Efficiently Finding Augmenting Paths

How long will it take Ford-Fulkerson to find max flow? That may depend partly on luck…

Example: In the graph below, any augmenting path crossing the **middle edge** will grow by **one unit at a time**. The **max number of augmentations** is $1000 + 1000 = 2000$. **That's a lot** for such a simple graph.







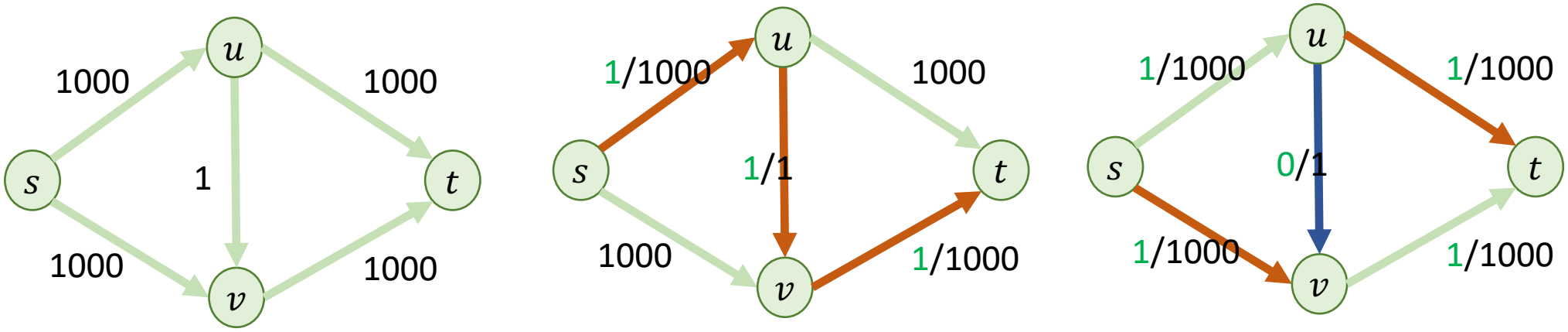
This keeps going on for 2000 steps…

Can we do anything to **guarantee better efficiency**?

Efficiently Finding Augmenting Paths

How long will it take Ford-Fulkerson to find max flow? That may depend partly on luck…

Example: In the graph below, any augmenting path crossing the **middle edge** will grow by **one unit at a time**. The **max number of augmentations** is $1000 + 1000 = 2000$. **That's a lot** for such a simple graph.



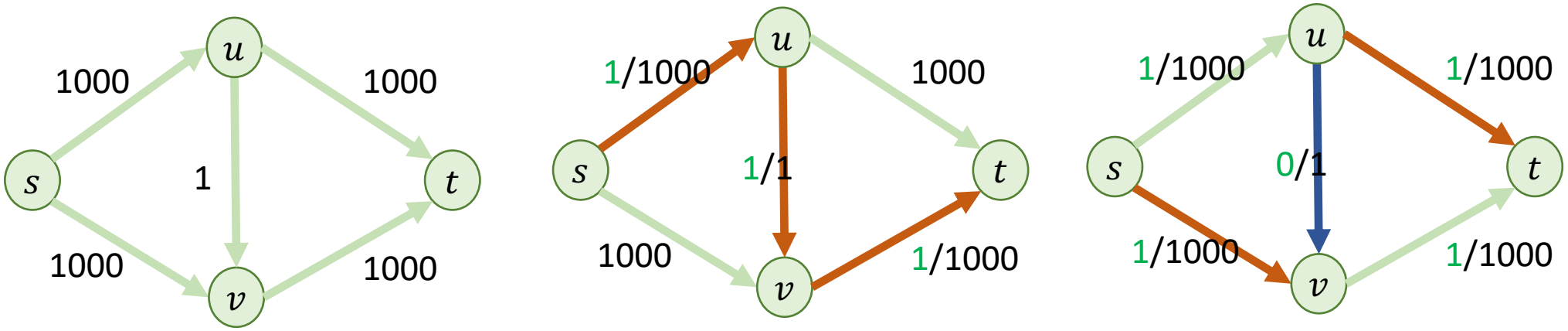
What if we know the **shortest augmenting path** from s to t?

- **Breadth first search** will find the shortest path (in terms of number of steps).
- The middle edge **cannot be part** of the shortest path from both sides – at most one of the two is shortest.
- If we find augmenting path using BFS, we are **guaranteed to not reuse** the same edge in this way.

Edmonds-Karp Runtime

If we use BFS to find augmenting paths for the Ford Fulkerson method, we call it the *Edmonds-Karp Algorithm*.

Any **iteration of Edmonds-Karp** will perform a BFS and update the flow in $\boldsymbol{O(m)}$ **time**.



Every iteration of Edmonds-Karp will **fully saturate** at least one of m edges.

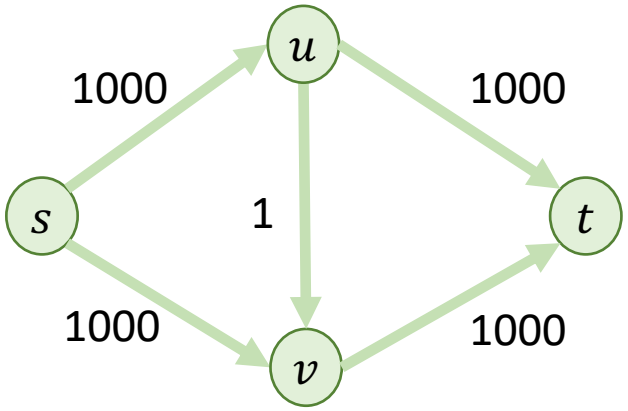
- If two augmenting paths are the **same length**, one will **not undo** an edge of the other.
- There are a total of $\boldsymbol{n-1}$ **possible path lengths**, which BFS considers in order.

Result: $O(nm^2)$ runtime.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



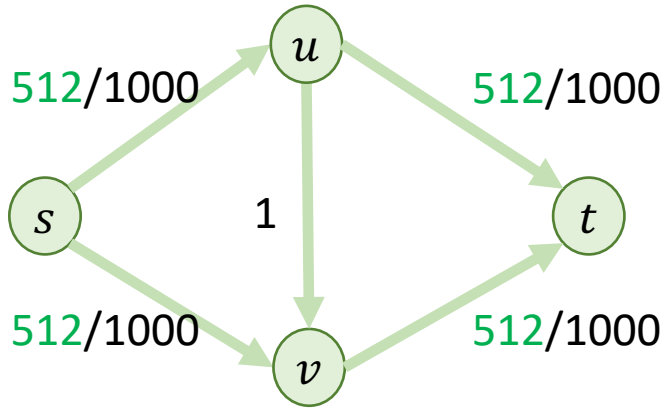
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 512$

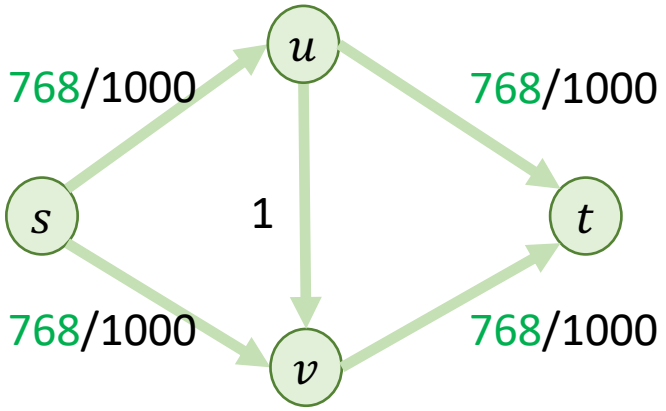
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 256$

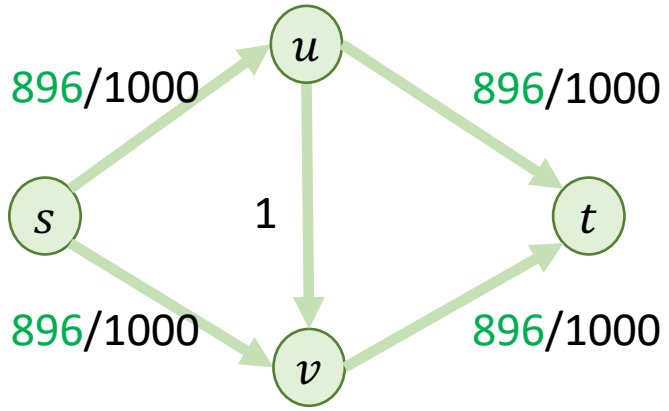
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 128$

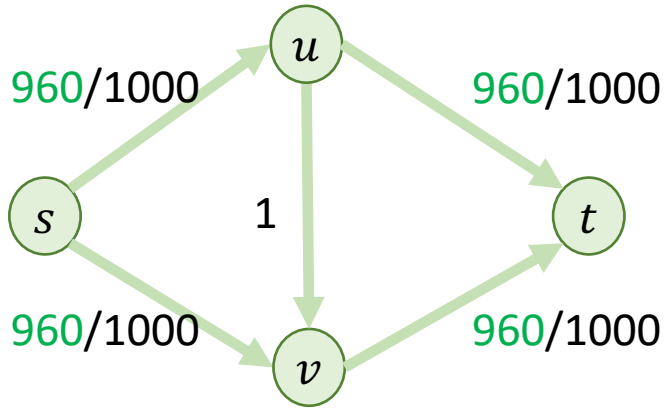
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 64$

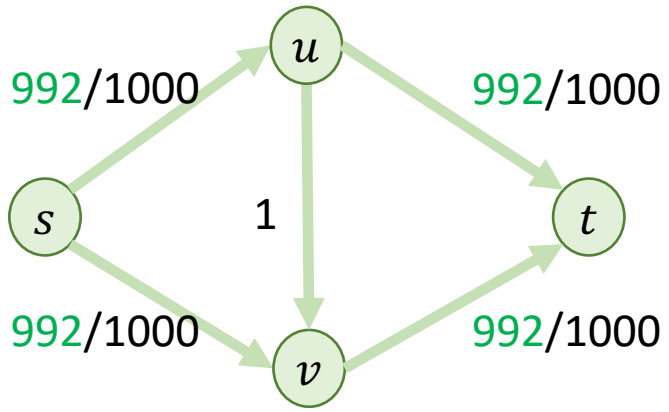
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 32$

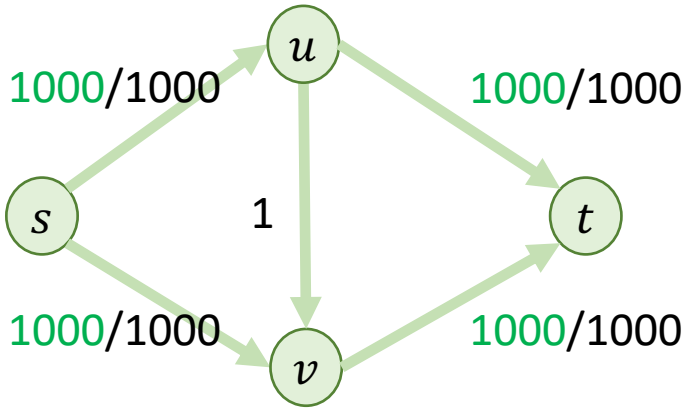
Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ some edge.

Another Efficient Approach: Capacity Scaling

Premise: Look for higher-capacity augmenting paths first.

- Pick a **high value Δ**. Only look for paths whose **capacity is $\geq \Delta$**.
- If there are no such paths, **make Δ smaller** and try again.
- By the time $\Delta = 1$, we will find **all possible augmenting paths**.



$\Delta = 8$

Proposal: Make Δ a **power of 2**, and **divide by 2** when necessary.

- We can **compose any number** as a sum of powers of 2.
- The initial Δ can be the **largest power of 2** that is $\leq$ source capacity.

Runtime:

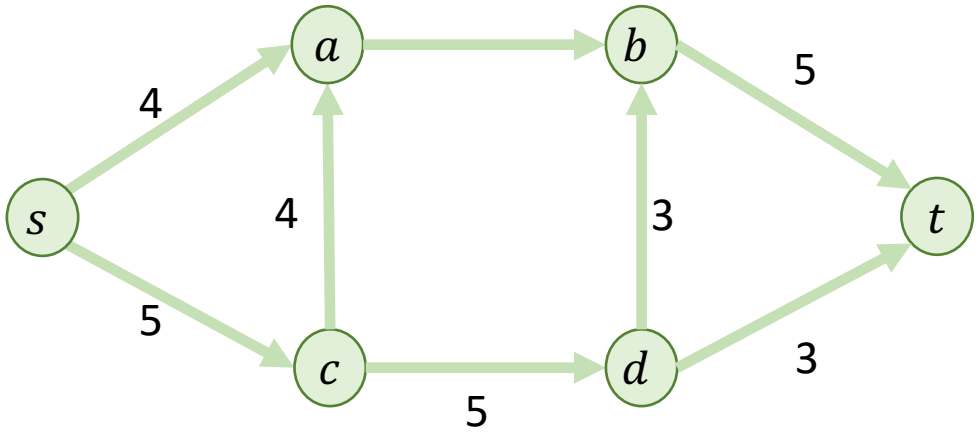
- As before, each **augmentation is $O(m)$** due to search costs.
- For each Δ, at most m **augmentations** can happen.
  - Each edge can be increased at most once if we use BFS.
- There are $O(lg\ C)$ **values of Δ** to check, where C is the capacity.

Overall: $O(m^2 lg\ C)$

Unlimited Capacity

Some edges might not have practical limits to the flow.

- How would this be represented in a flow network?

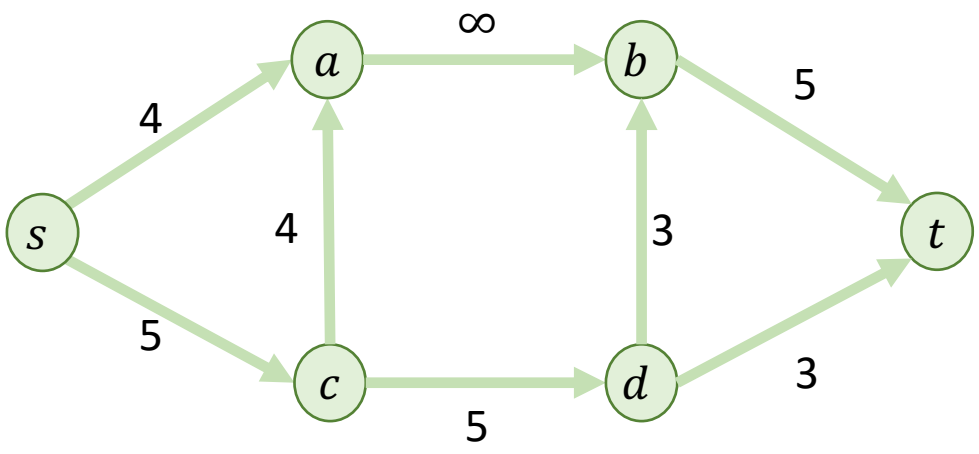


Example: Suppose there is **no limit** to the flow **from a to b**.

Unlimited Capacity

Some edges might not have practical limits to the flow.

- How would this be represented in a flow network?



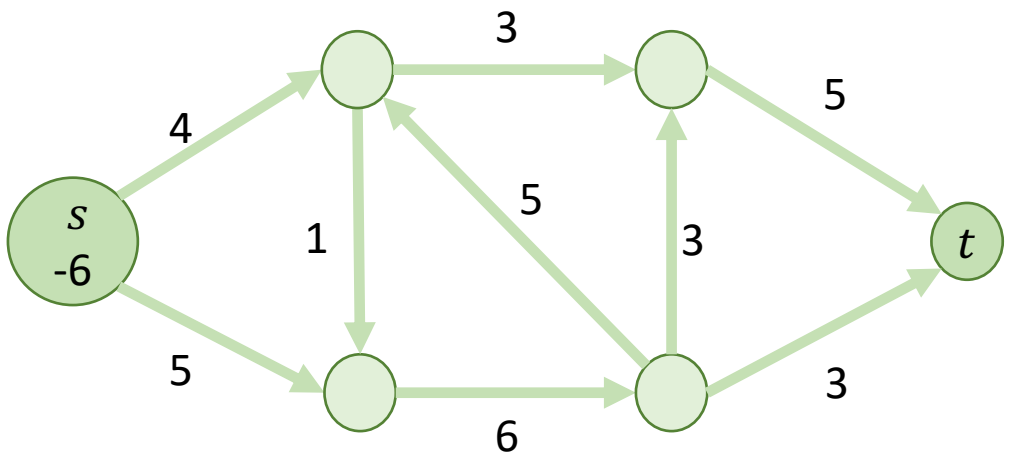
Example: Suppose there is **no limit** to the flow **from a to b**.

Solution: Simply label the capacity as **infinity**.

Producer Nodes

We may want to add simple, practical constraints to our flow network.

Imagine that our source has a **limited production capacity**.
- We could denote that with a **negative capacity value** on the source itself.



Example: The graph on the left is different from a basic flow network because it has a **source limit** of 6.

No more than 6 units of flow can be produced by the source.

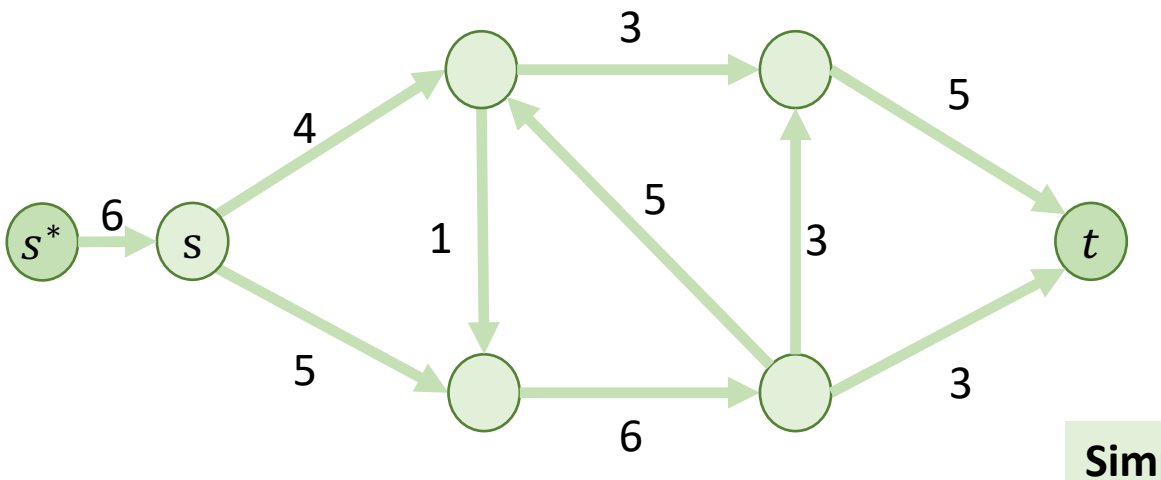
We know how to solve an ordinary **max flow**.
- Can we **use what we know** to solve this problem?

Producer Nodes

We may want to add simple, practical constraints to our flow network.

Imagine that our source has a **limited production capacity**.

- We could denote that with a **negative capacity value** on the source itself.



Simple solution: Create a new source.

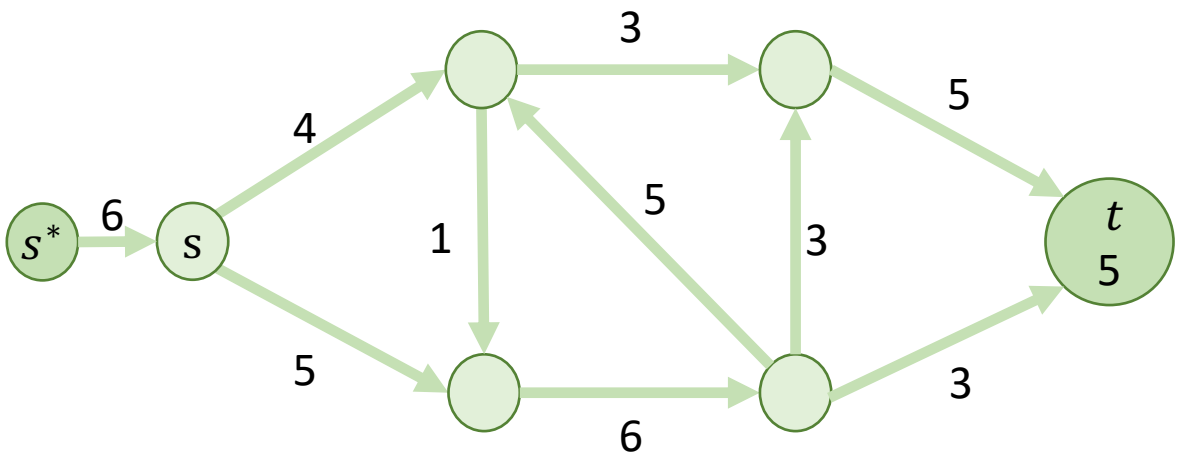
- We can only reach s from s^* by a path with **bounded capacity**.

Consumer Nodes

What if our sink has limits to what it **can consume**?.

Imagine that our sink has a **limited consumption capacity**.

- We could denote that with a **capacity value** on the sink itself.

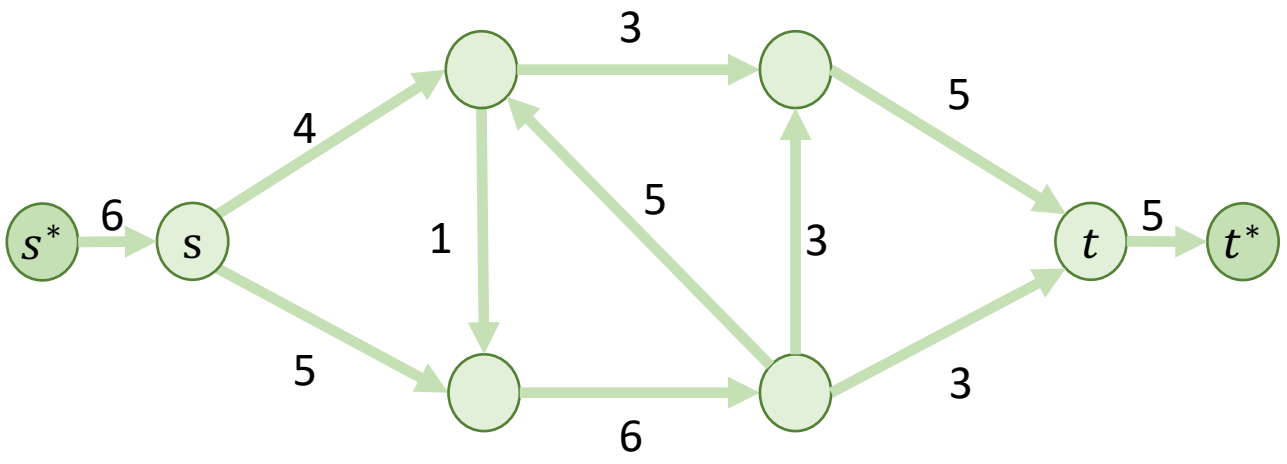


Consumer Nodes

What if our sink has limits to what it **can consume**?.

Imagine that our sink has a **limited consumption capacity**.

- We could denote that with a **capacity value** on the sink itself.



As with the source, we can restrict the sink by **creating a new sink**.

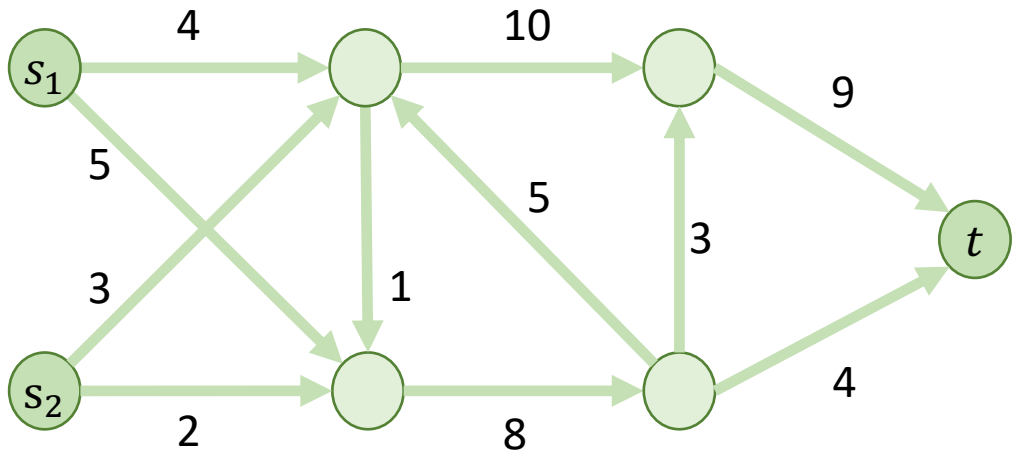
- The **only way to reach** the new sink is via a **capacity-limited edge**.

Dual Sources

Why would we need to limit capacity at both **source and sink**?

- Wouldn't the result be the **same either way**?

Ah, but consider what happens if we now had **two different sources**.



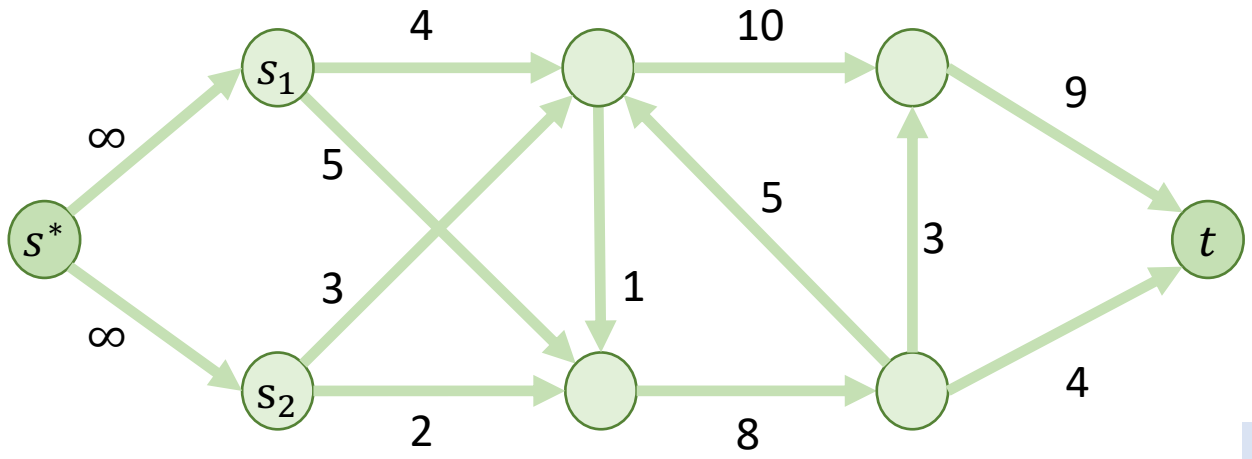
We would still like to solve this using **techniques we already know**.

Dual Sources

Why would we need to limit capacity at both **source and sink**?

- Wouldn't the result be the **same either way**?

Ah, but consider what happens if we now had **two different sources**.



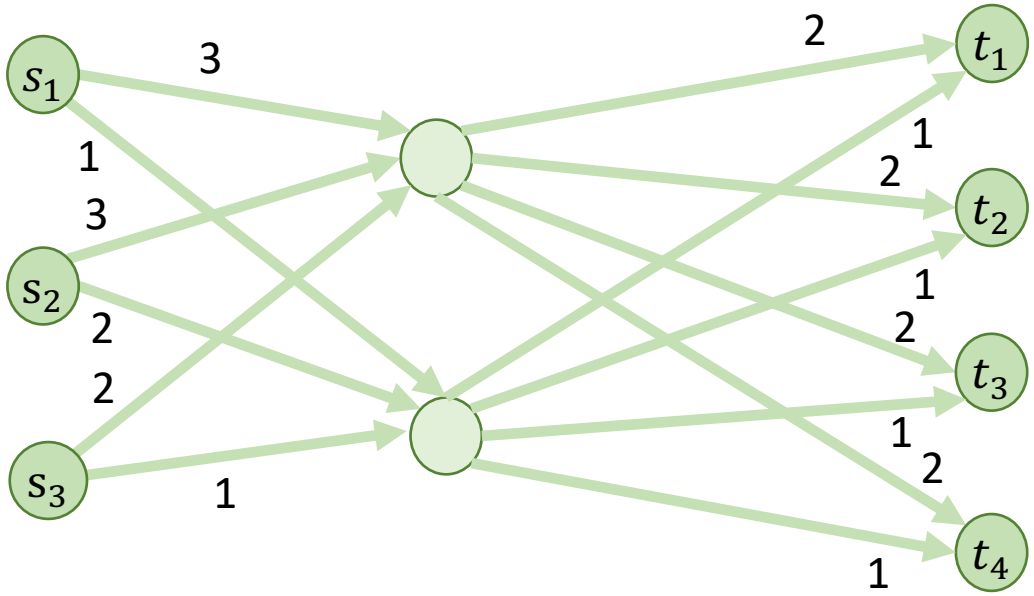
Solution: Feed **both sources** from a new **single source**.

We do not know how much flow each source will put out, so we can use **infinite capacity lines** to feed each source.

Multiple Sources and Sinks

Can we generalize situation where we have **many sources and sinks**?

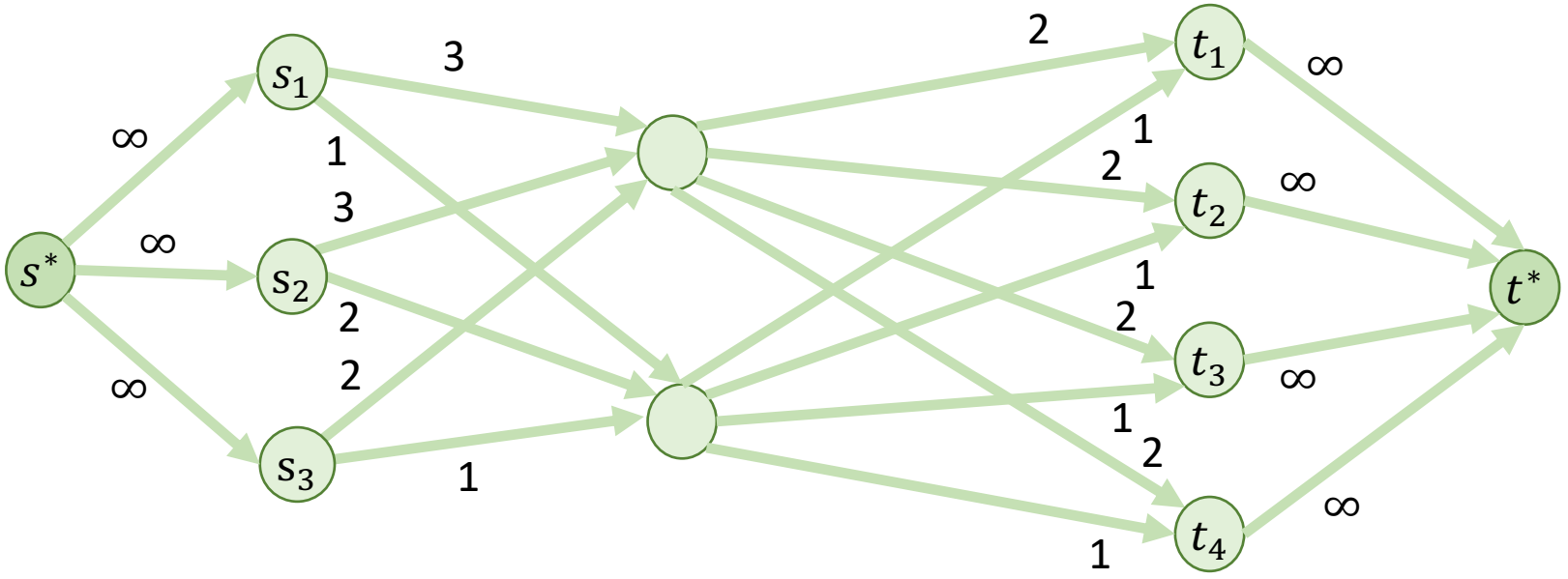
Example: Three different sources and four sinks.



Multiple Sources and Sinks

Can we generalize situation where we have **many sources and sinks**?

Example: Three different sources and four sinks.

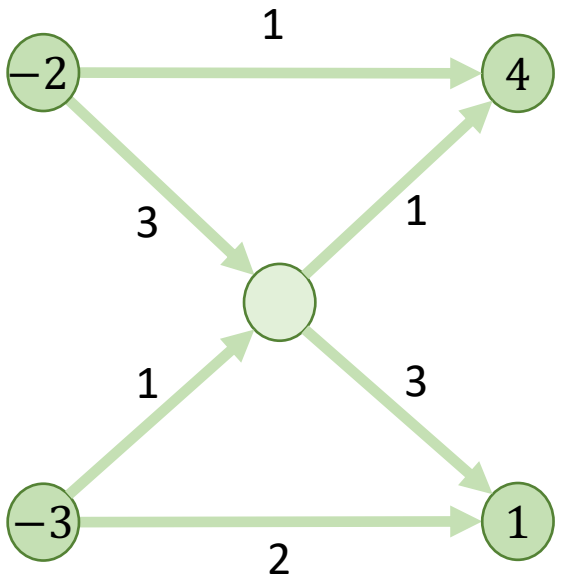


The key is to be sure we end up with **exactly one source** and **exactly one sink**.

Producers and Consumers (circulations)

Suppose we have a network in which:

- Some nodes produce flow. (negative demand, $d_v < 0$)
- Some nodes consume flow. (positive demand, $d_v > 0$)

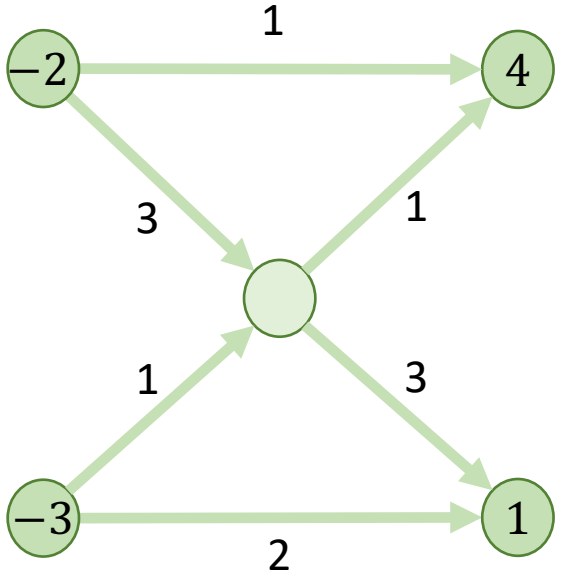


Example: How can we arrange this graph into something we can solve as a **max flow**?

Producers and Consumers (circulations)

Suppose we have a network in which:

- Some nodes produce flow. (negative demand, $d_v < 0$)
- Some nodes consume flow. (positive demand, $d_v > 0$)
- All nodes should be "satisfied": producers send out their full value, consumers receive their full value. $f^{out}(v) - f^{in}(v) = d_v$.

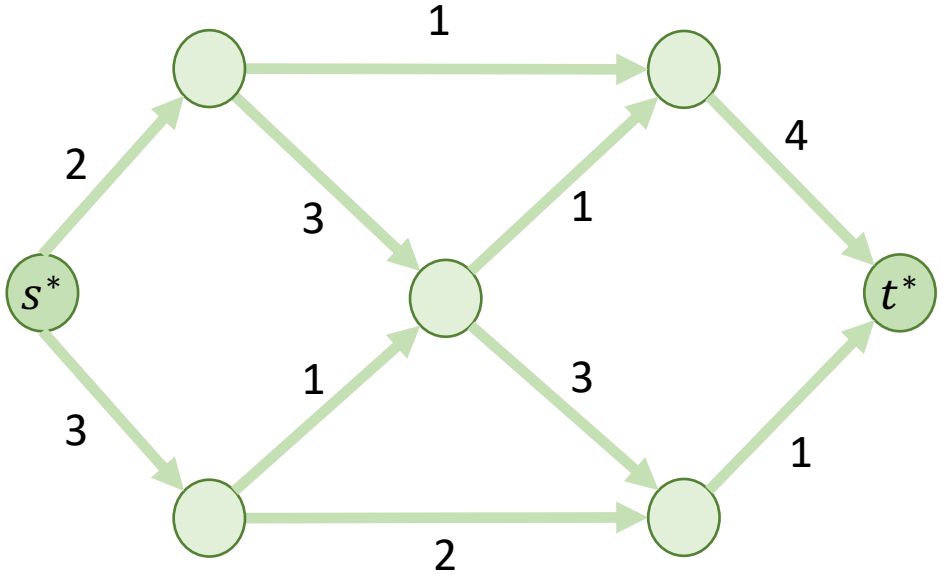


Example: How can we arrange this graph into something we can solve as a **max flow**?

Producers and Consumers (circulations)

Suppose we have a network in which:

- Some nodes produce flow. (negative demand, $d_v < 0$)
- Some nodes consume flow. (positive demand, $d_v > 0$)
- All nodes should be "satisfied": producers send out their full value, consumers receive their full value. $f^{out}(v) - f^{in}(v) = d_v$.

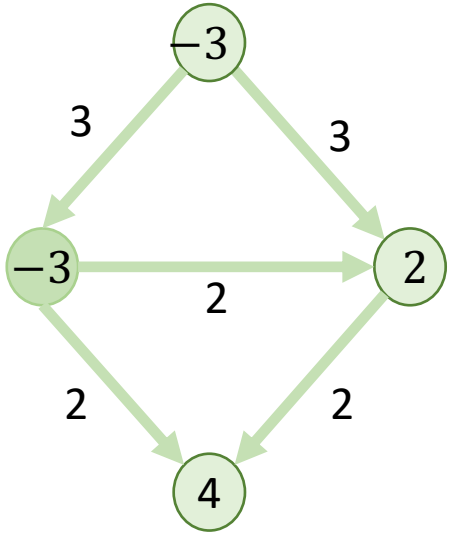


Solution:

- New source **with edges to each producer**.
  - Edges from source to producer reflects producer capacity.
- New sink **with edges from each consumer**.
  - Edges from consumer to sink reflects sink capacity.
- Feasible solution if max flow = Σ positive demands $= \Sigma$ negative demands

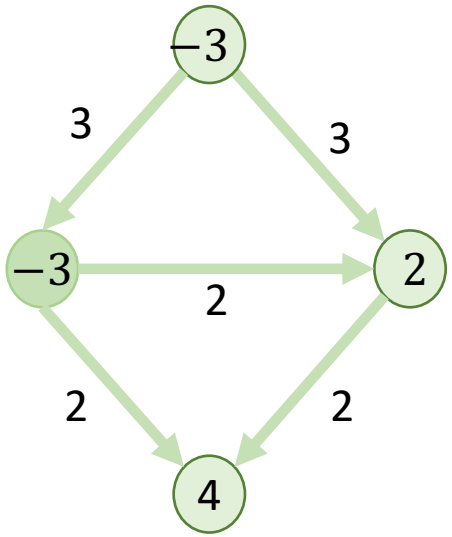
Producers and Consumers (circulations)

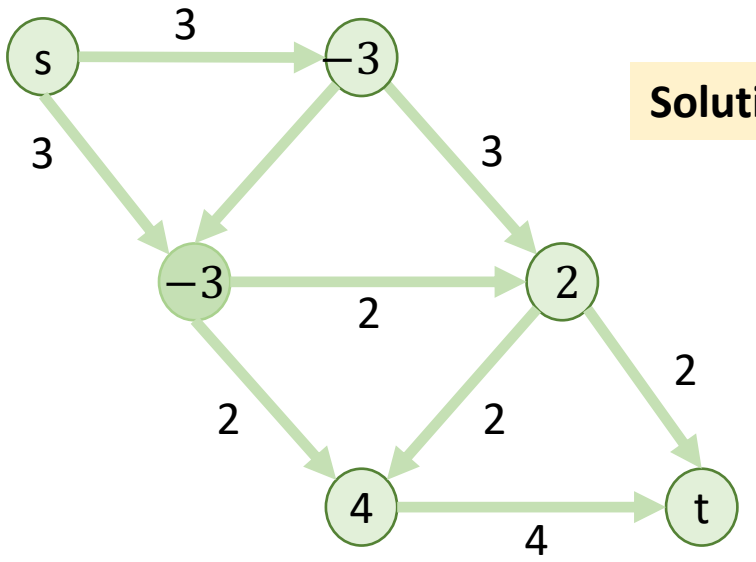
A consumer might also be an internal node



Producers and Consumers (circulations)

A consumer might also be an internal node

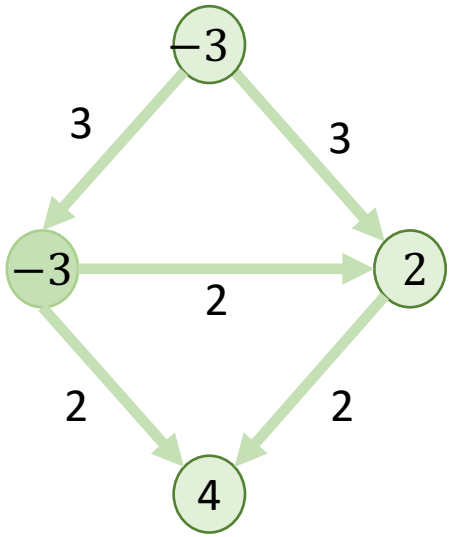


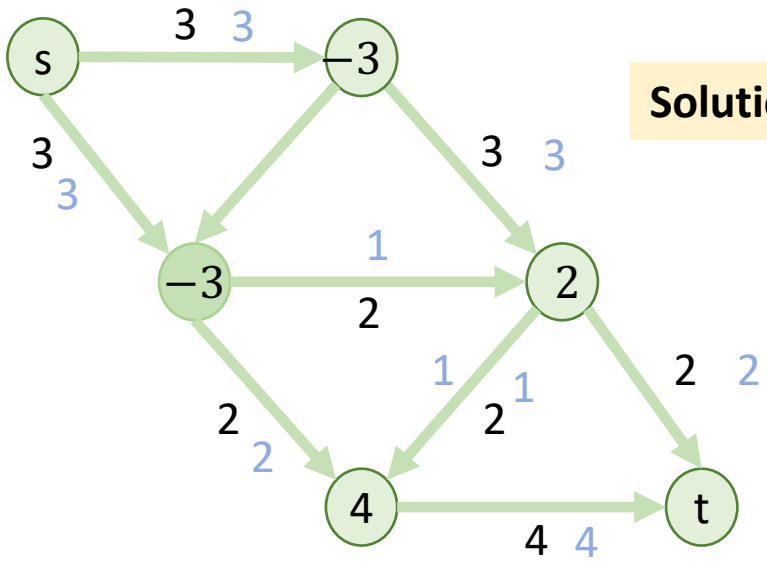


Solution: Find a max flow of 6.

Producers and Consumers (circulations)

A consumer might also be an internal node



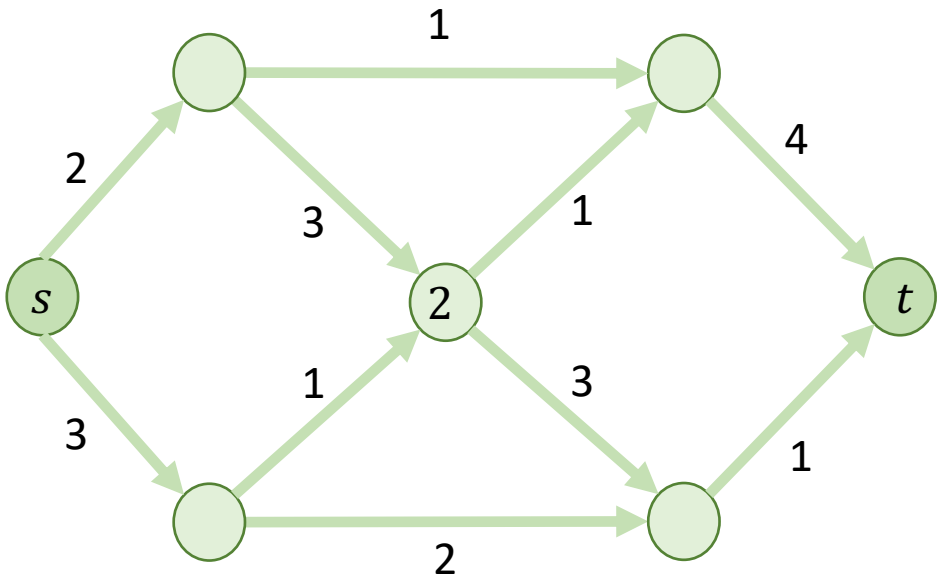


Solution: Find a max flow of 6.

Limited Capacity Nodes

Suppose that some node by itself has capacity limits?:

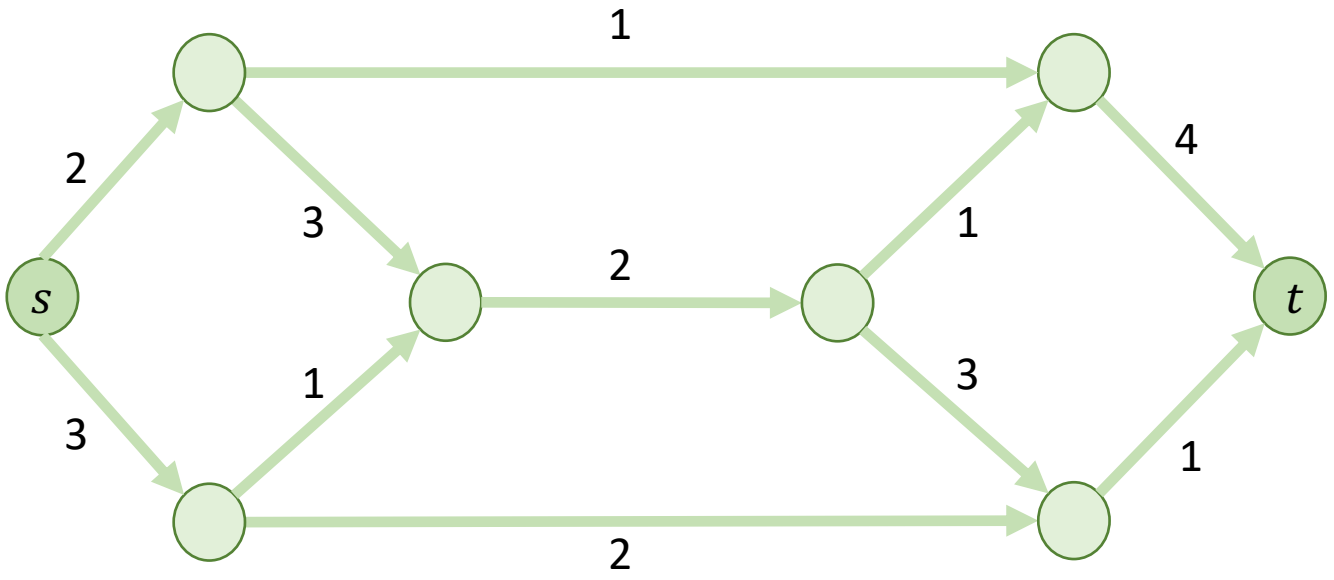
Example: Suppose that only 2 units of flow can flow across the middle.



Limited Capacity Nodes

Suppose that some node by itself has capacity limits?:

Example: Suppose that only 2 units of flow can flow across the middle.



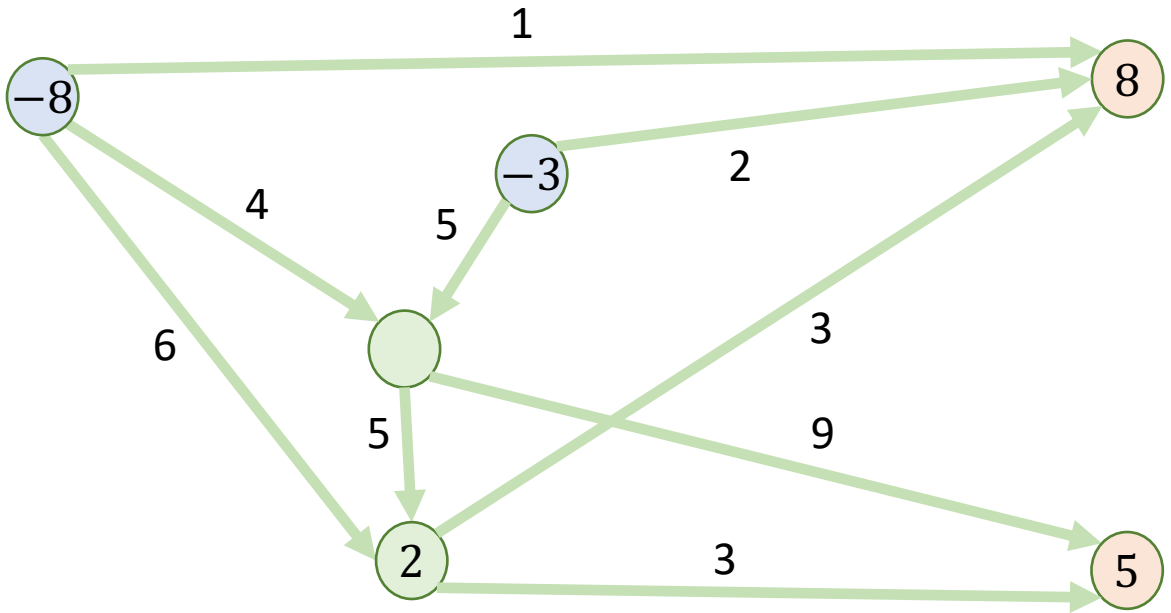
Solution: Turn **one node into two** connected by an edge. Let the edge represent the **capacity limit**.

Practice

Rewrite the following graph as a standard flow network.

Find the **max flow**.

The blue nodes are capacity-limited **producers**.
The brown nodes are **consumers**.

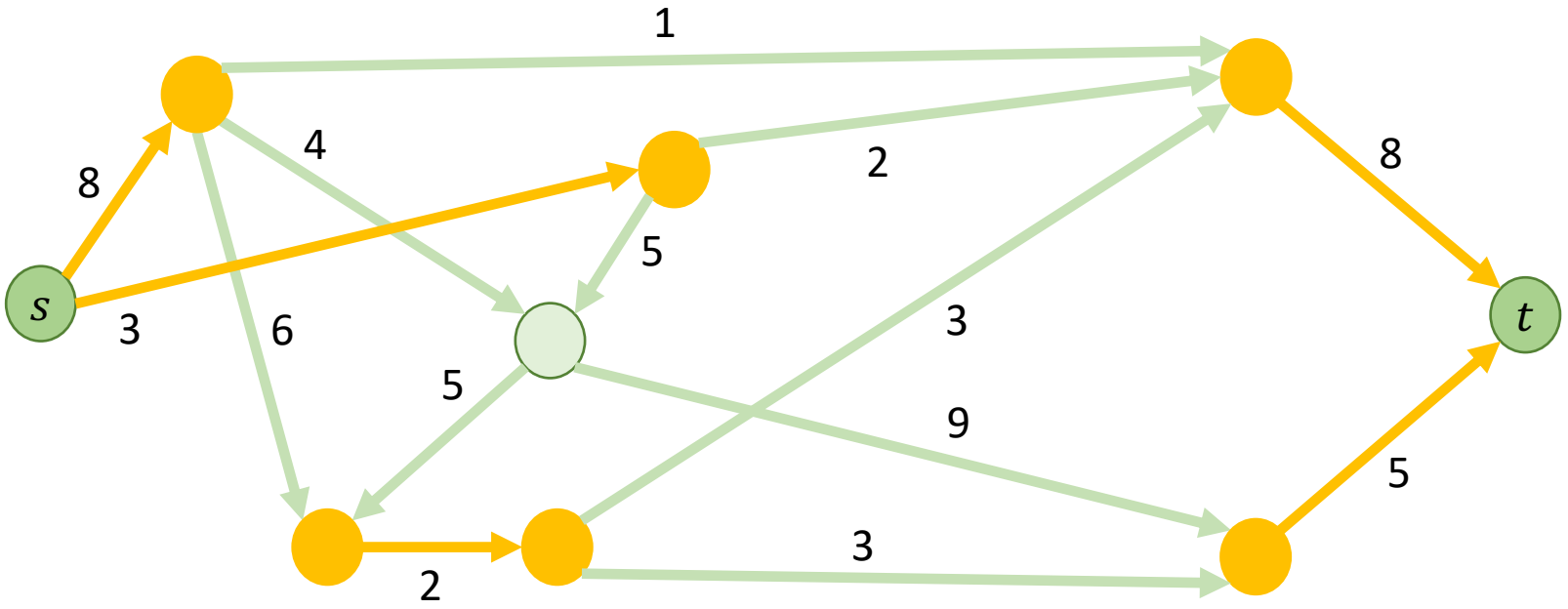


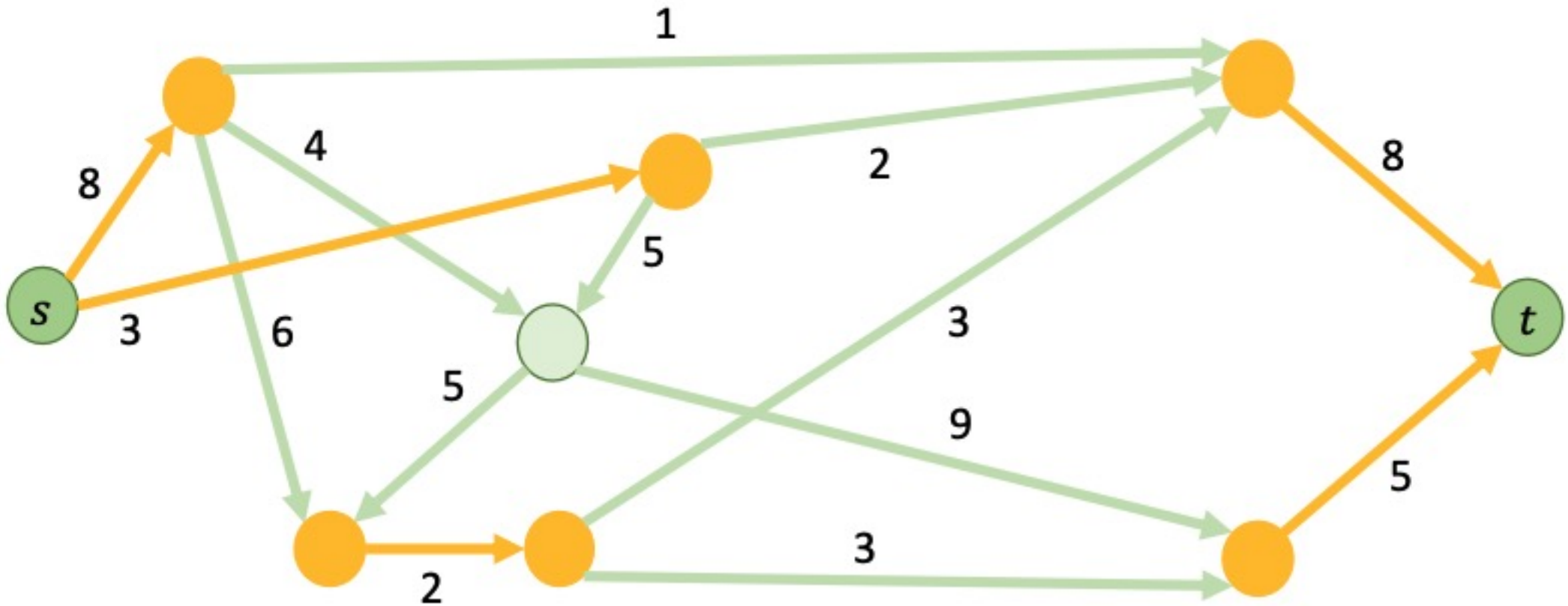
Practice

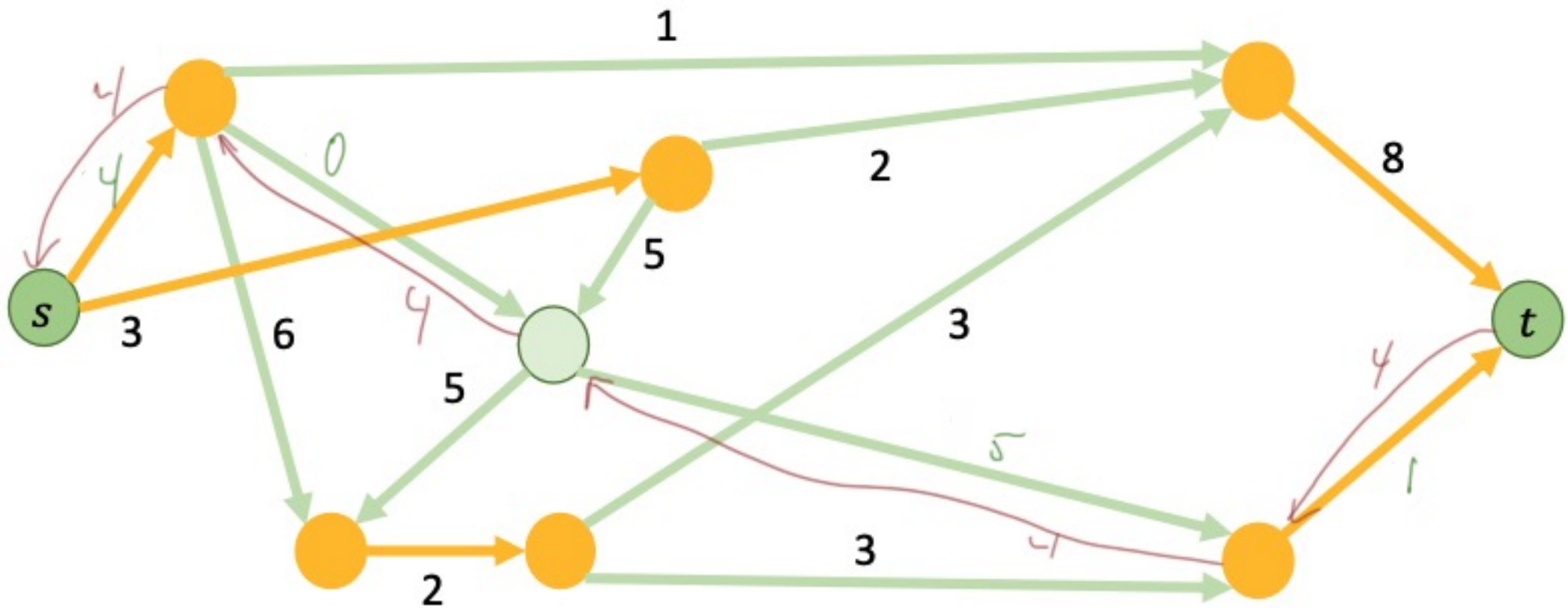
Rewrite the following graph as a standard flow network.

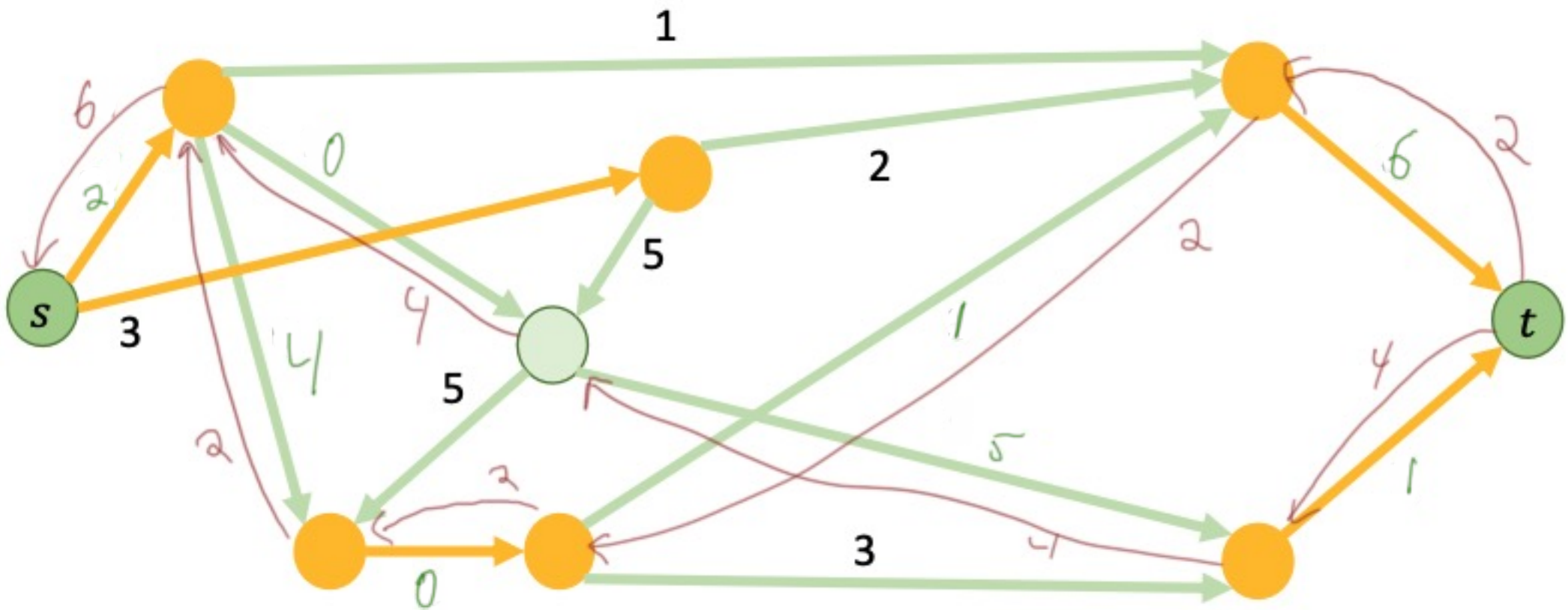
Find the **max flow**.

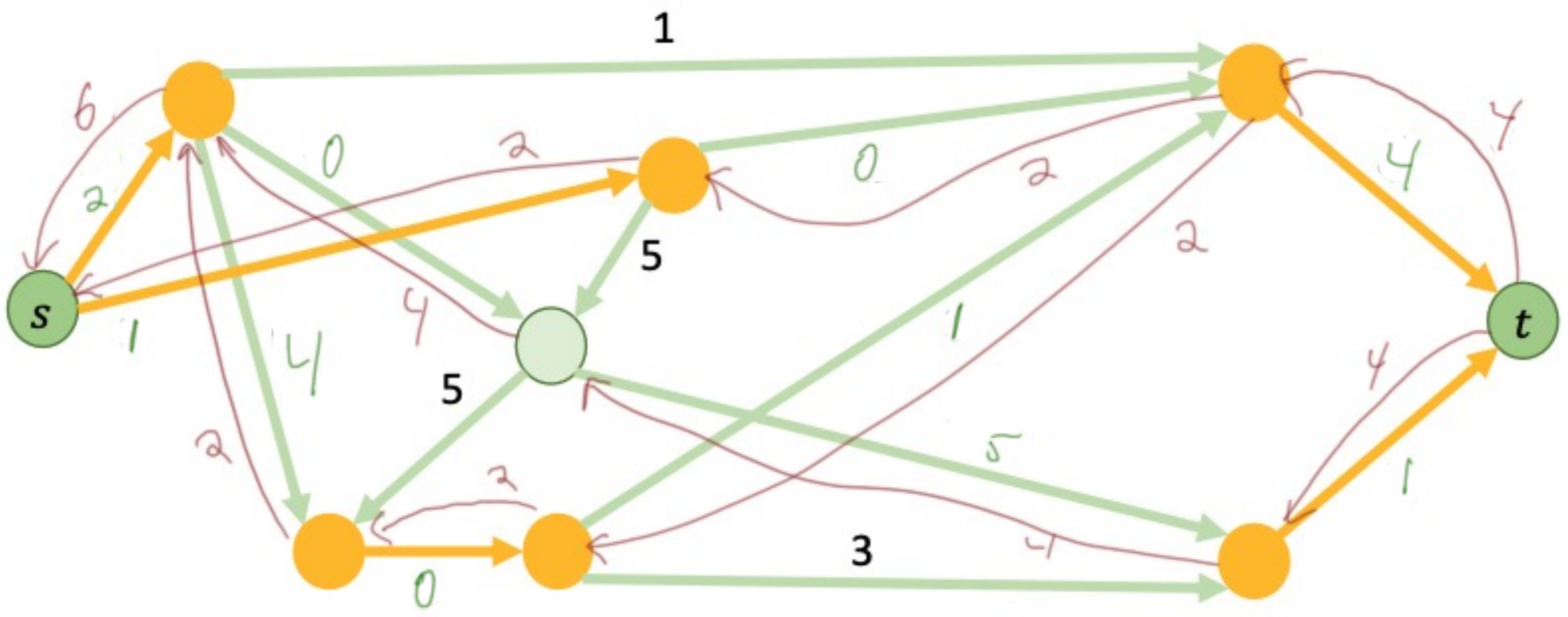
The blue nodes are capacity-limited **producers**.
The brown nodes are **consumers**.



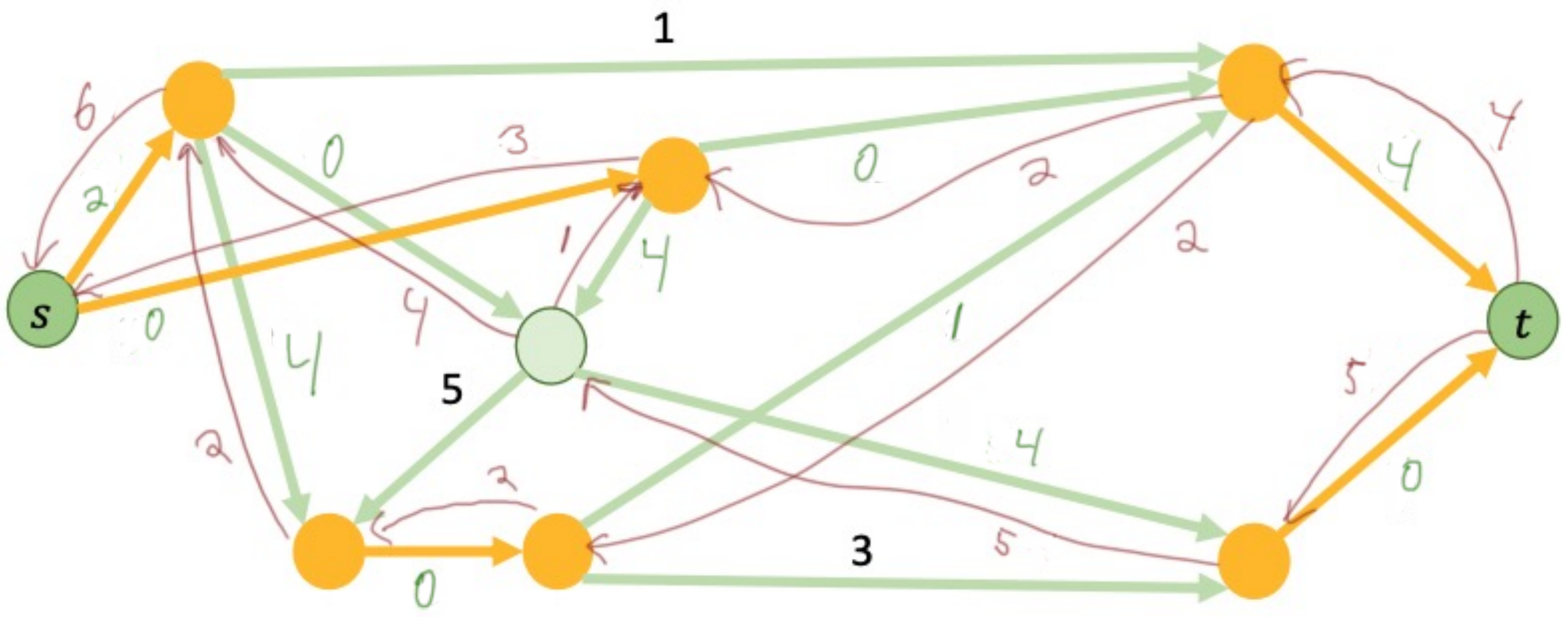

1
8
4
2
8
5
s
3
6
3
5
9
t
5
3
2

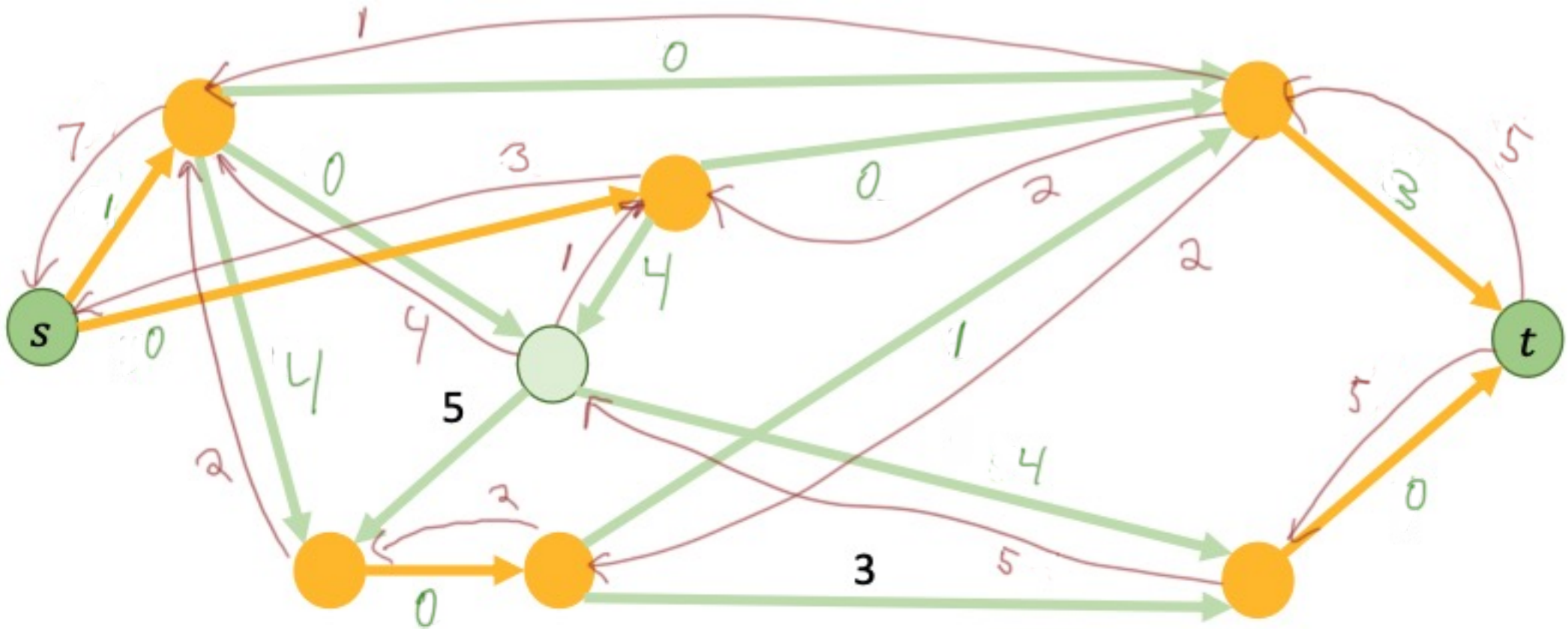

1
4
4
0
2
8
5
s
3
6
4
3
t
5
4
5
1
4
2
3

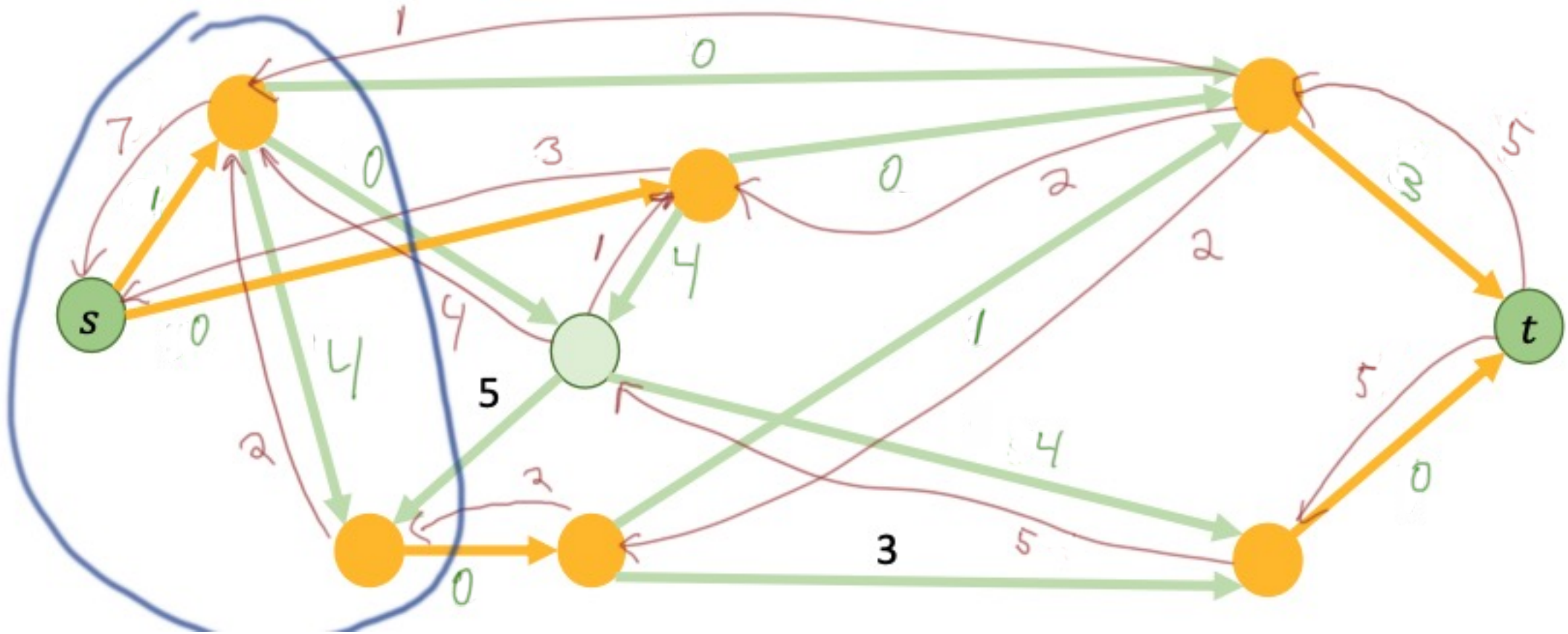

1
6
2
0
2
5
s
3
4
4
5
2
1
6
2
t
4
5
1
2
3
4
0
3

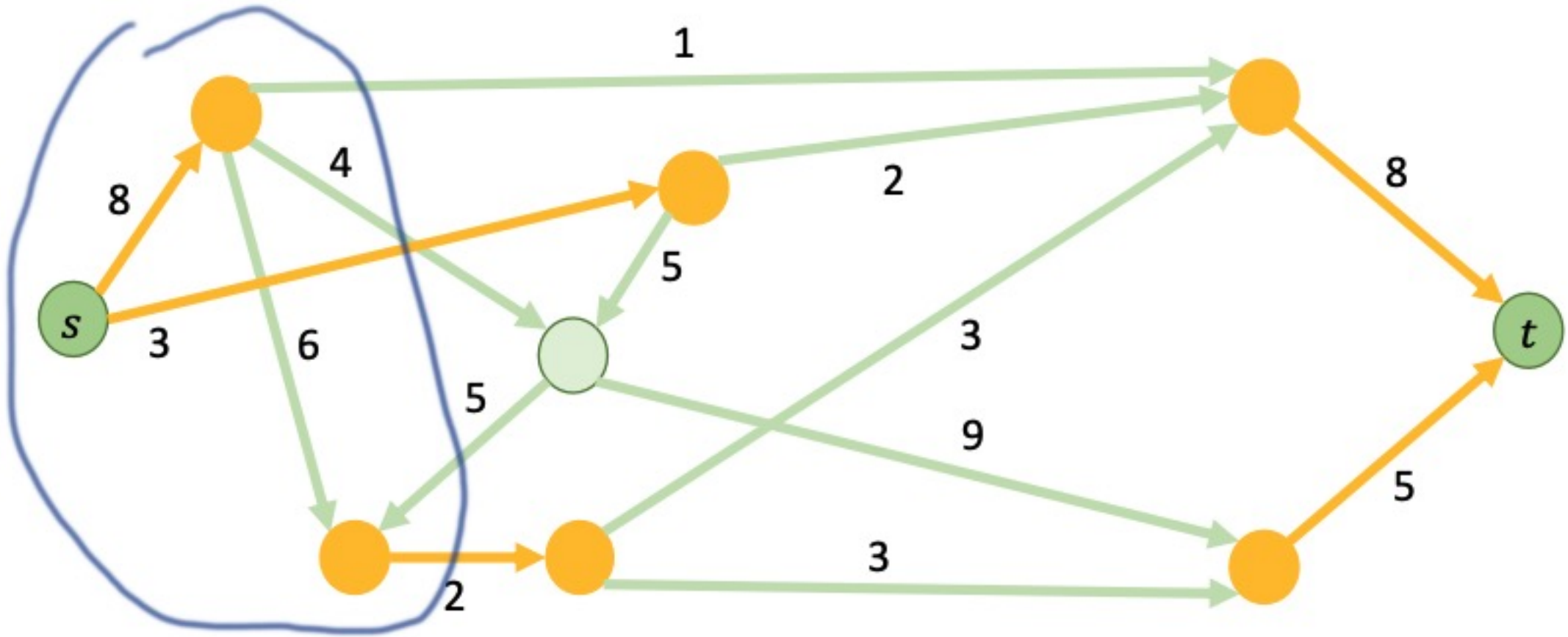


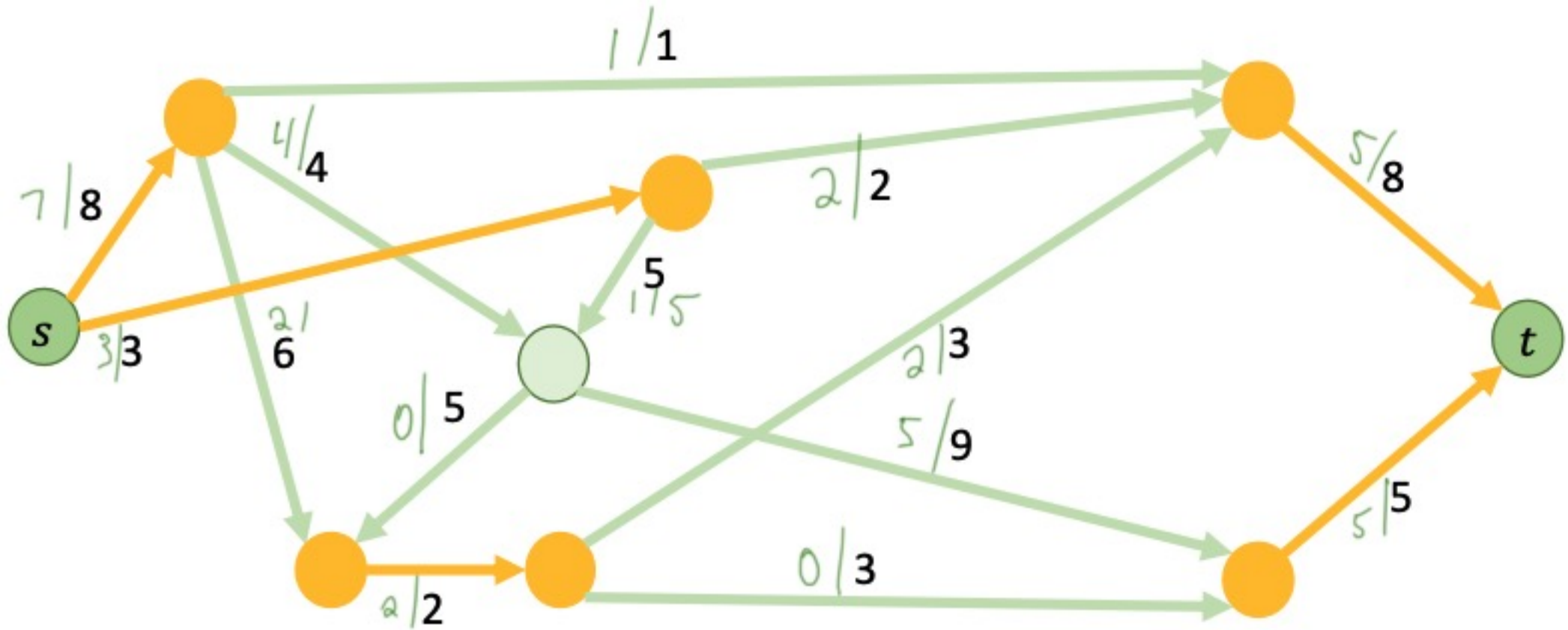
1
5
5
3
s
t


1
s
t
5
3
6
2
0
3
0
2
4
4
1
4
4
2
0
2
1
4
2
2
0
5
4
5
0
0


s
t
5
3


s
t
5
3


1
8
4
2
8
5
s
3
6
3
t
5
9
5
3
2


1/1
4/4
7/8
2/2
5/8
5
1/5
s
3/3
2/6
t
2/3
0/5
5/9
5/5
0/3
2/2

Flow Graphs for Task Assignment

Let’s consider a practical use of max flow problems.

Goal: We want to **assign workers to various tasks**.

- Everyone can be assigned a maximum (for example, 2) tasks.
- Some people are capable of performing more than one type of task.
- Not everybody knows how to do every kind of task.

Flow Graphs for Task Assignment

Let’s consider a practical use of max flow problems.

Goal: We want to **assign workers to various tasks**.

- Everyone can be assigned a maximum (for example, 2) tasks.
- Some people are capable of performing more than one type of task.
- Not everybody knows how to do every kind of task.

We can represent this relationship with a **bipartite graph**.

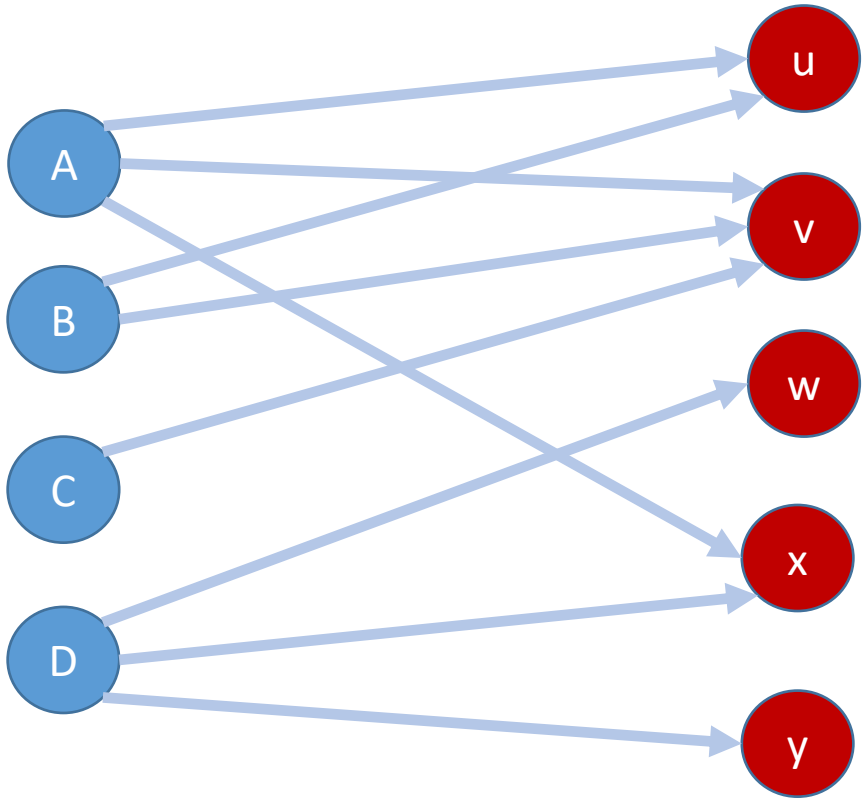
A bipartite graph is a graph which has **two groups** of nodes.

- Edges can **only cross** from one group to the other, not two nodes in one group.
- **Example**: One group can be a group of workers, the other a group of tasks.

Task Assignment, Example

The edges show **who can do which tasks**.

Workers



Tasks

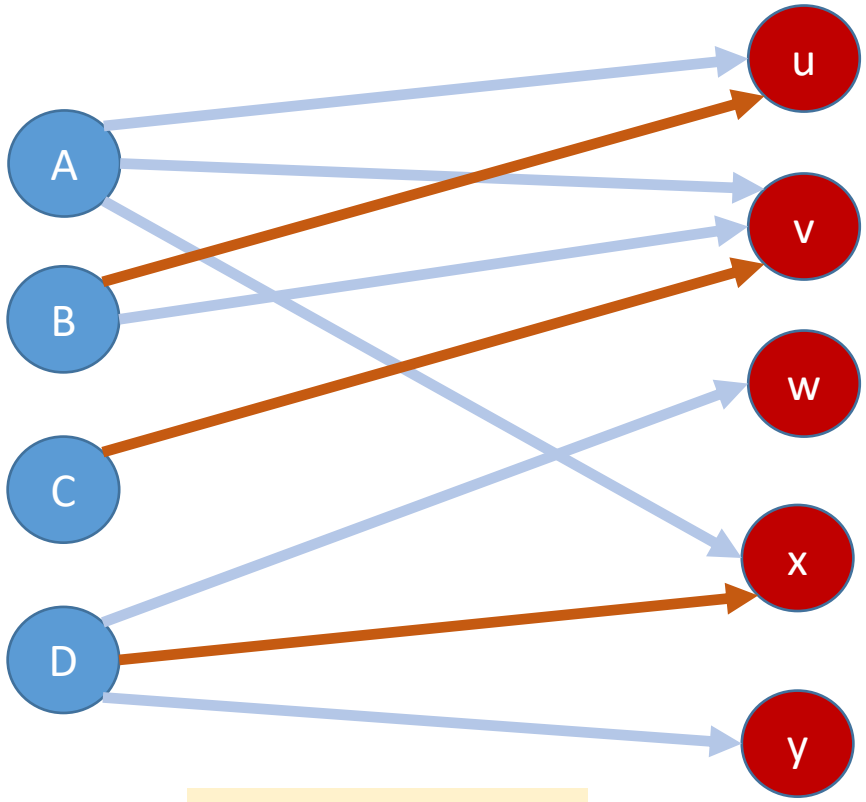
Goal: Find the **most tasks** that can get done.

Hint: this can easily be turned into a **flow problem**.

Task Assignment, Example

The edges show **who can do which tasks**.

Workers



Tasks

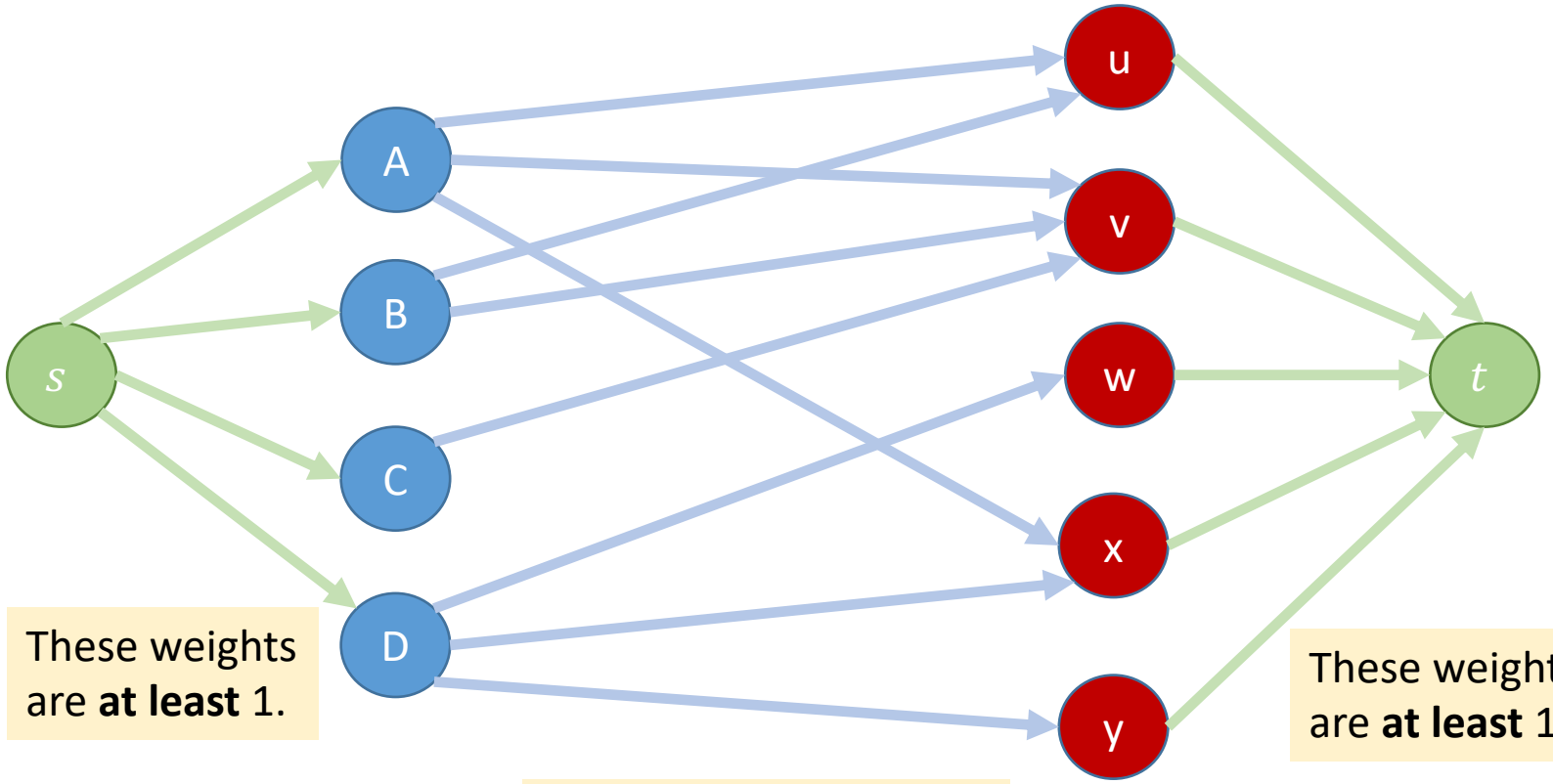
Goal: Find the **most tasks** that can get done.

Hint: this can easily be turned into a **flow problem**.

Sample solution
Can we do better?

Task Assignment, Example

The edges show **who can do which tasks**.



What would it mean if we use weights which are > 1?

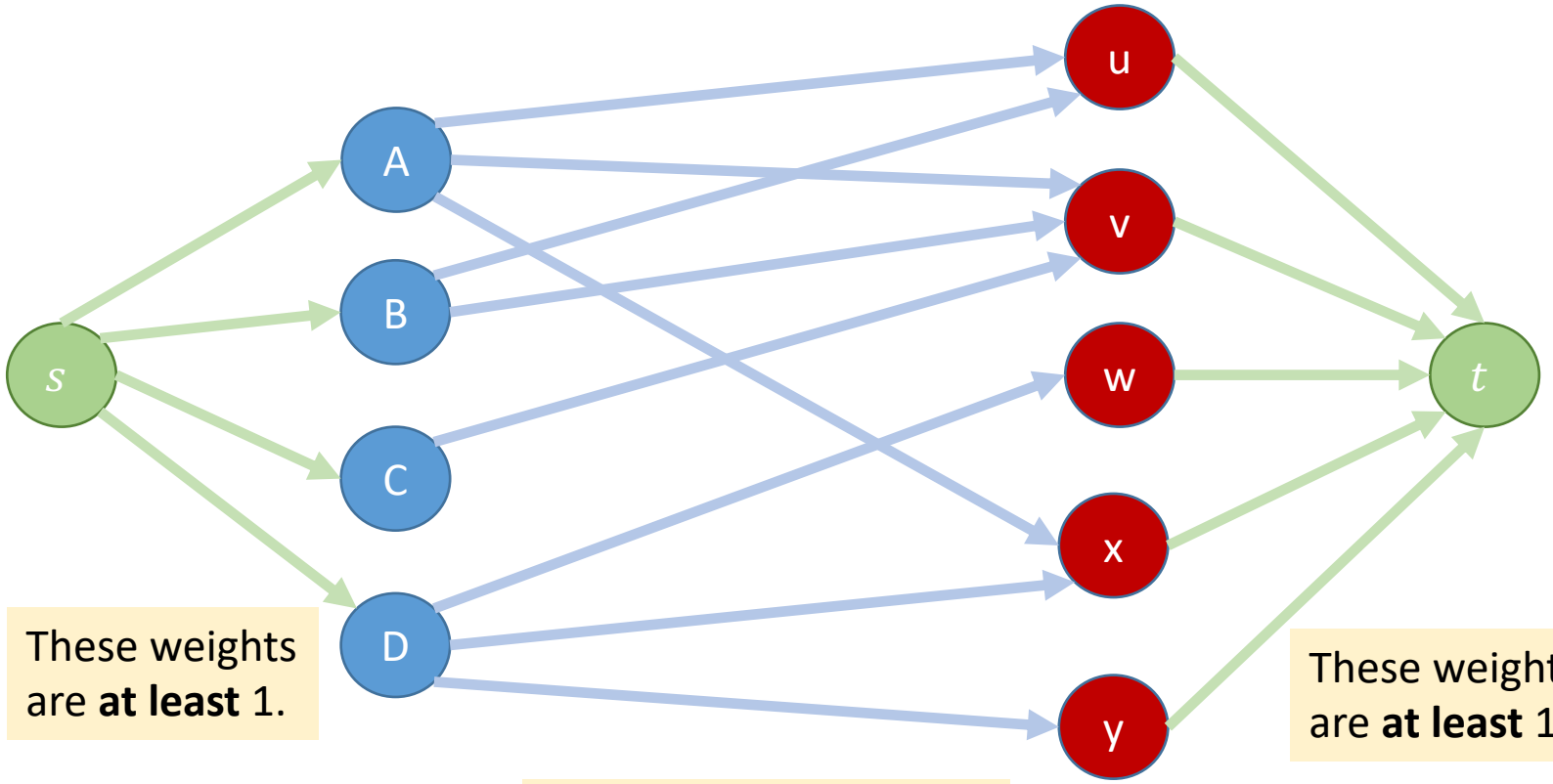
These weights are **at least** 1.

These weights are **at least** 1.

These weights are all 1.

Task Assignment, Example

The edges show **who can do which tasks**.



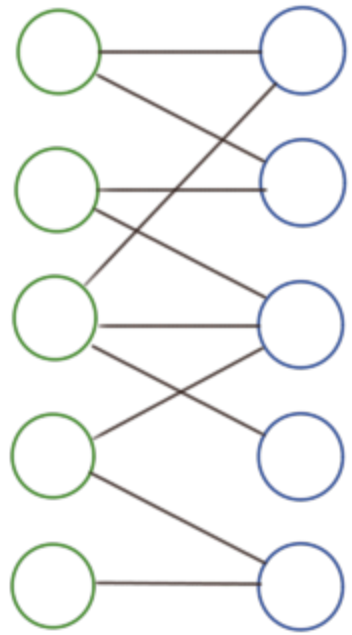
These weights are **at least** 1.

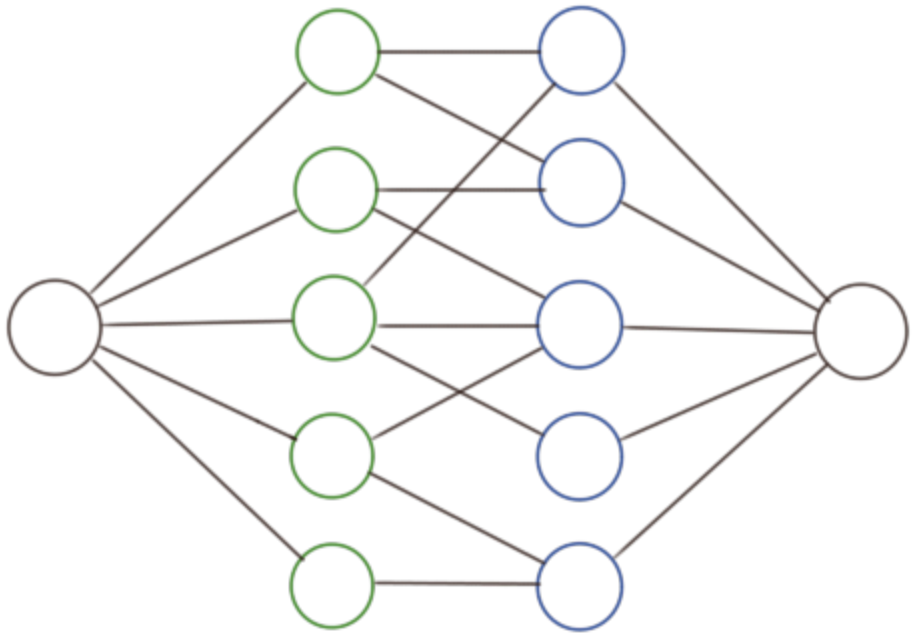
These weights are all 1.

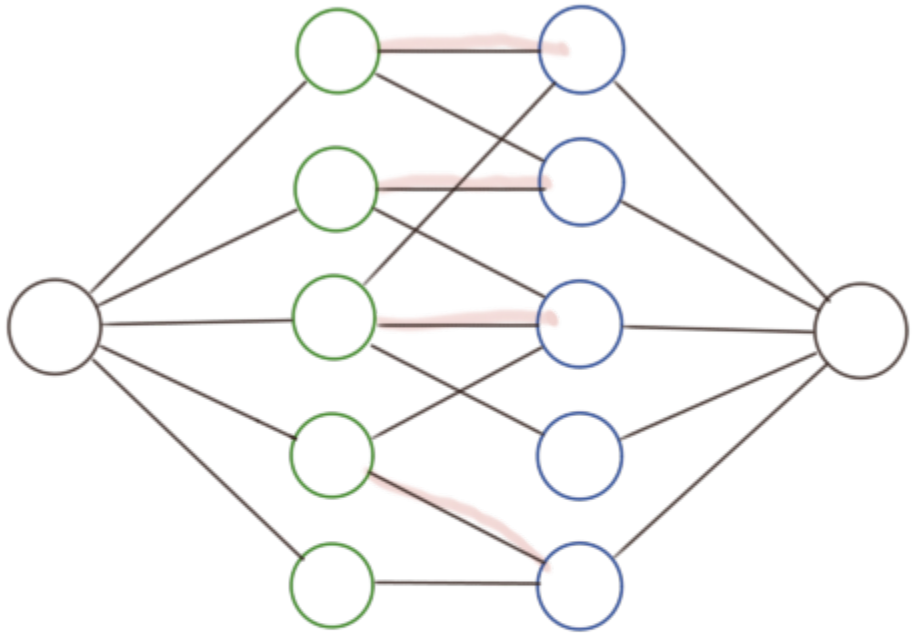
These weights are **at least** 1.

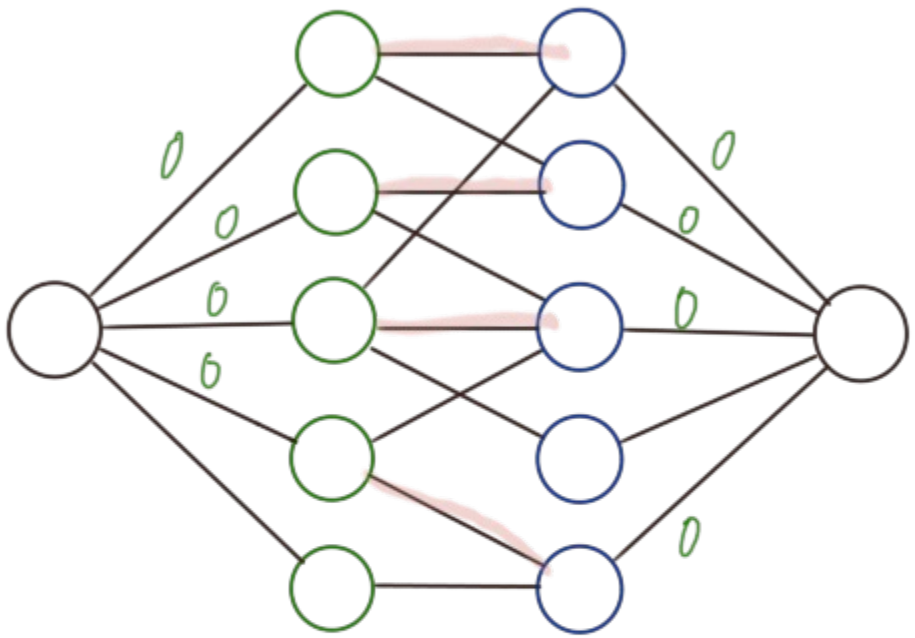
What would it mean if we use weights which are > 1?

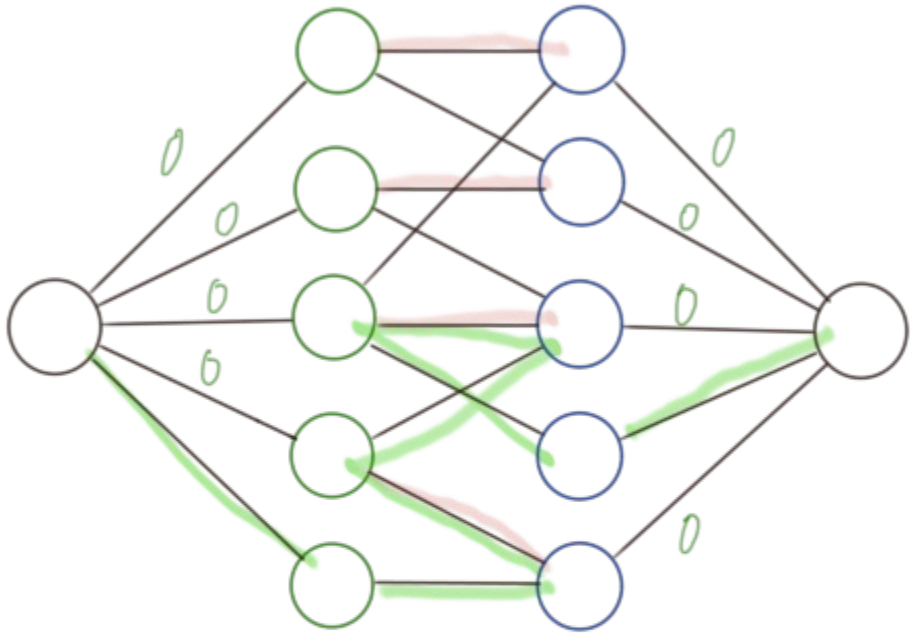
- If a worker can be assigned **more than one task** at once.
- If a task can be assigned to **multiple people** at once.

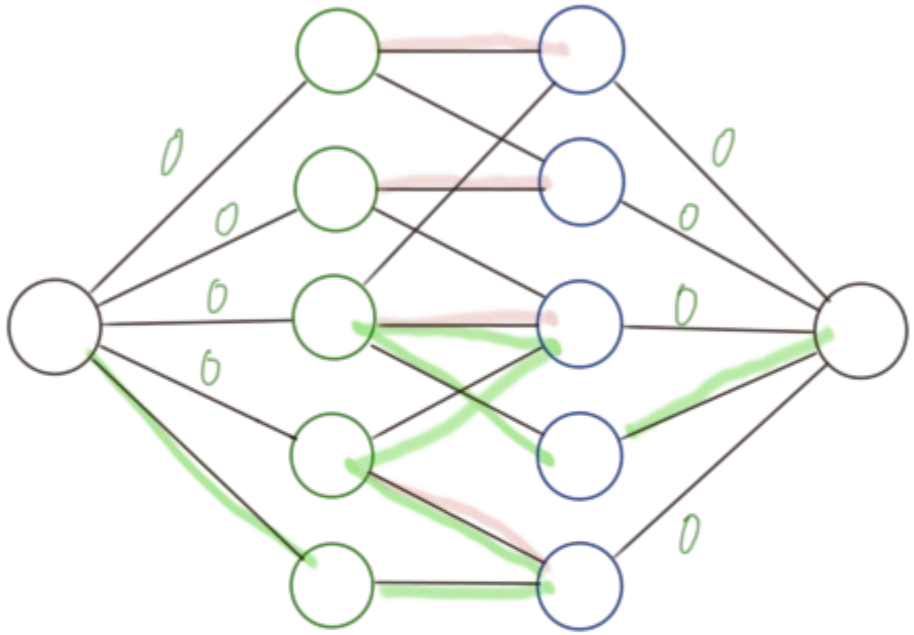


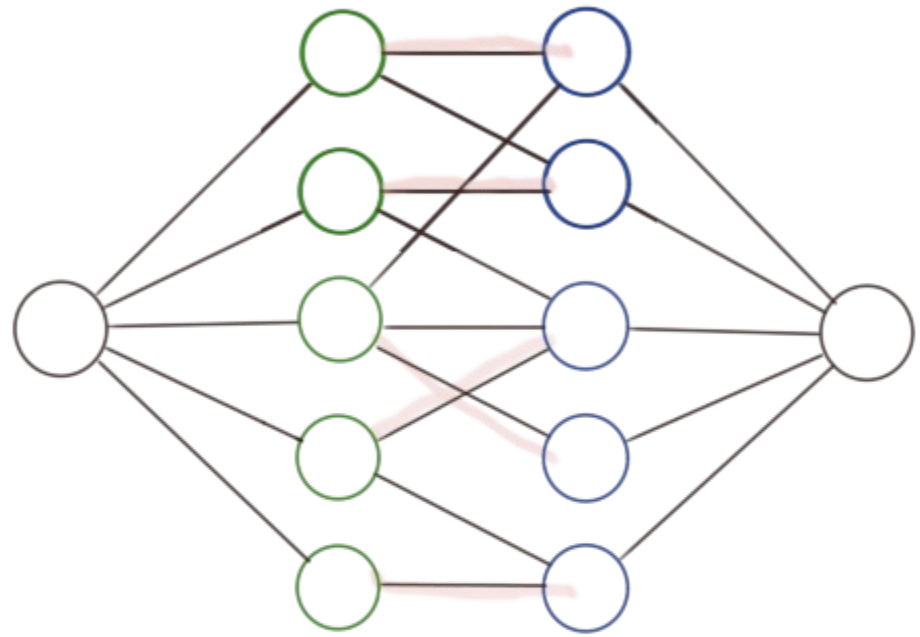


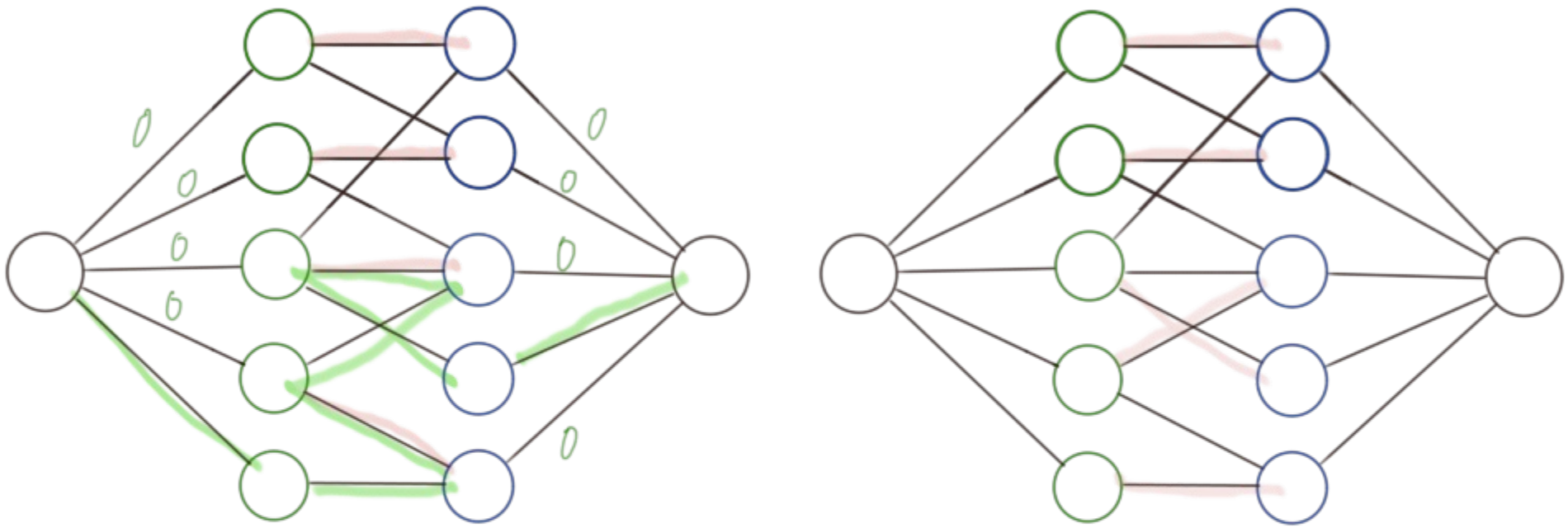







0
0
0
0
0
0
0





Any new path crosses the border

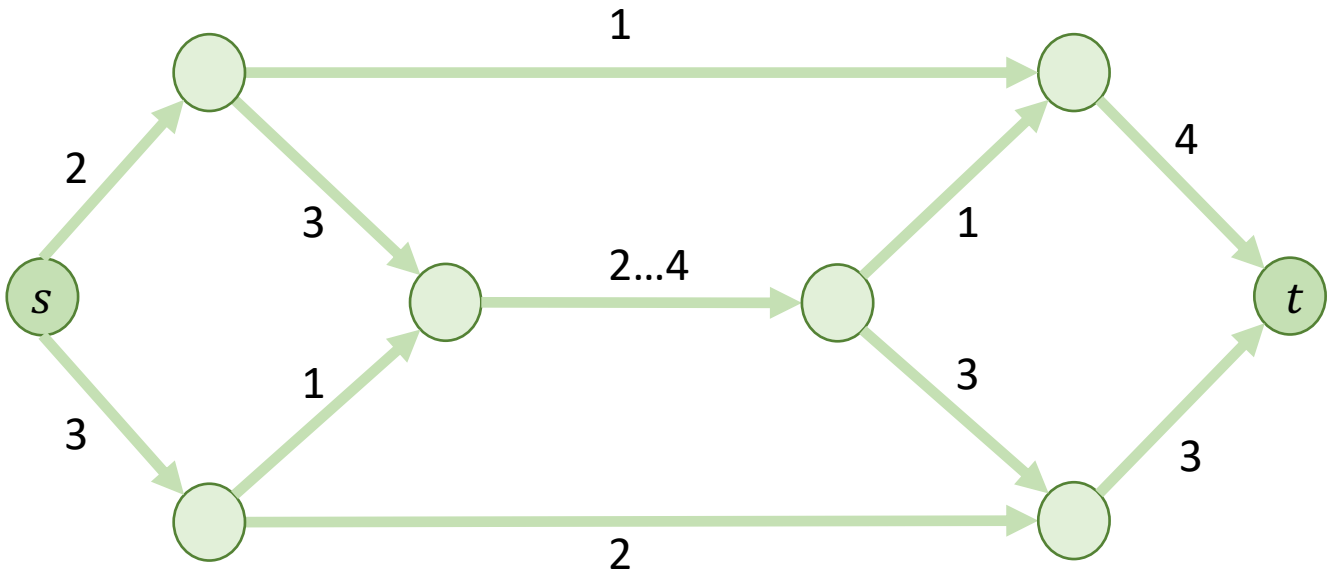
x times ->

x-1 times <-

Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

Example: The middle edge must have a flow of at least 2 and at most 4.

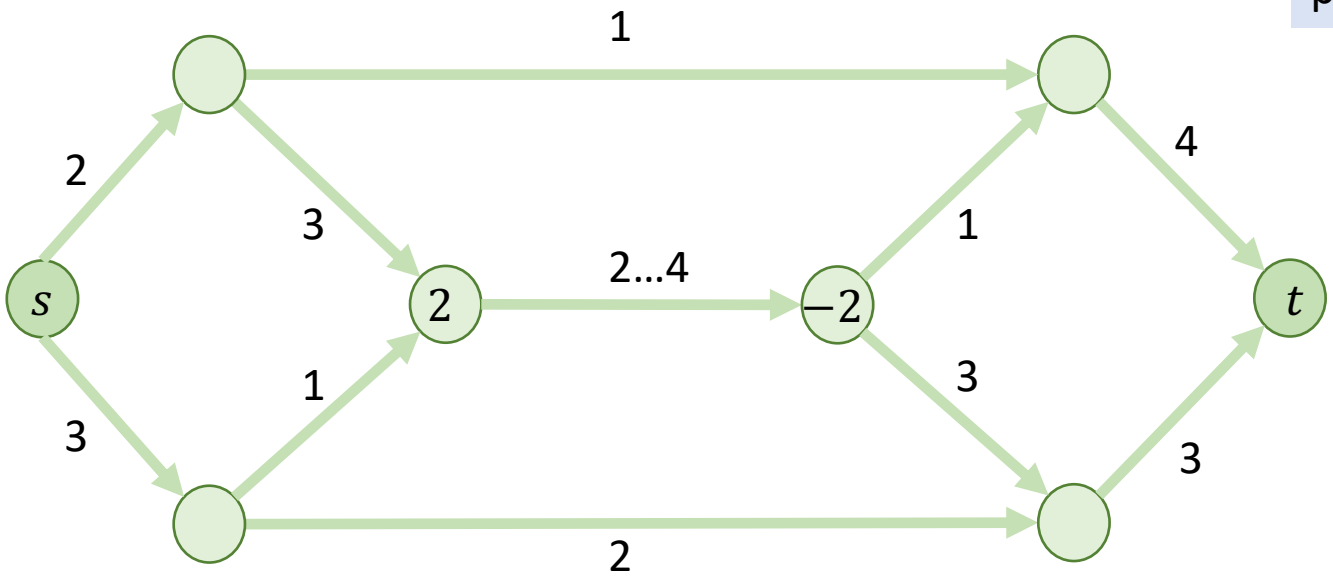


Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

Example: The middle edge must have a flow of at least 2 and at most 4.

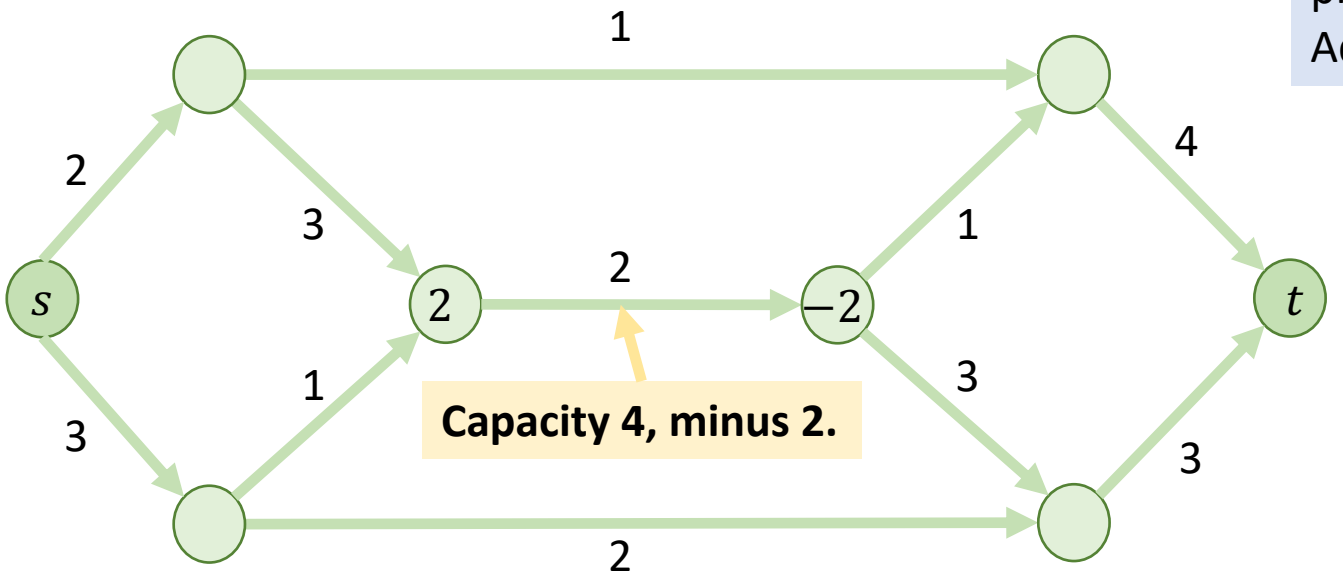
Impose a demand upstream, and produce the required amount downstream



Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

Example: The middle edge must have a flow of at least 2 and at most 4.

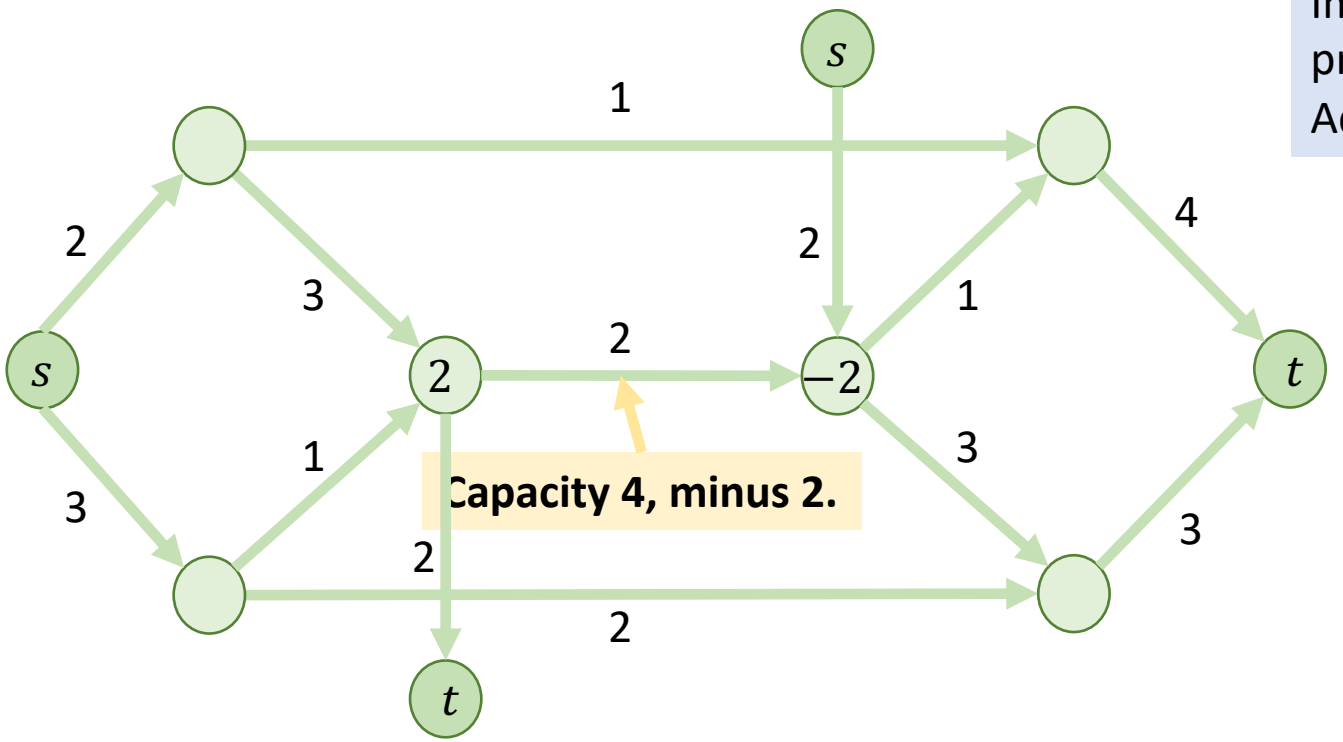


Impose a demand upstream, and produce the required amount downstream. Adjust the capacity to reserve space.

Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

Example: The middle edge must have a flow of at least 2 and at most 4.

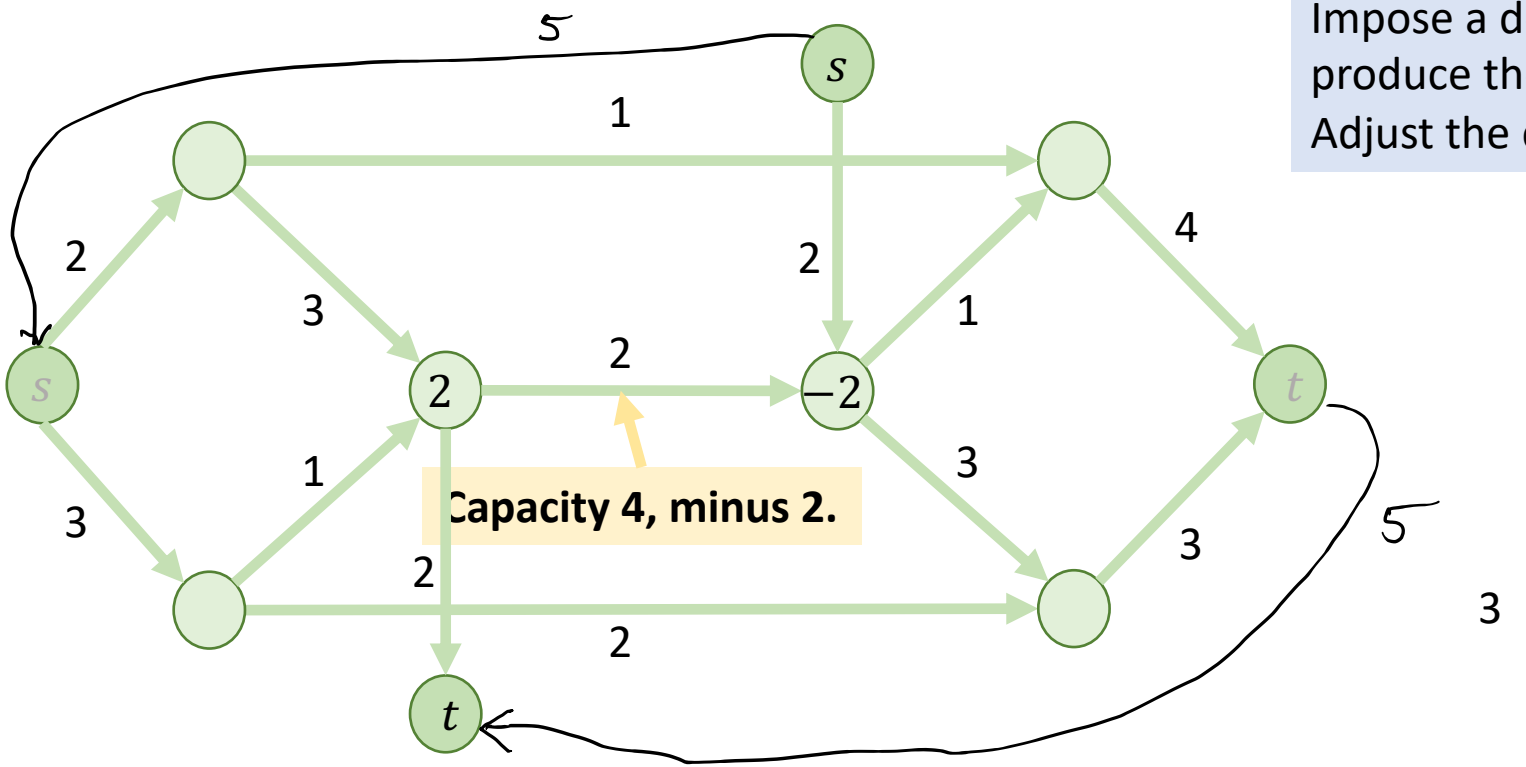


Impose a demand upstream, and produce the required amount downstream. Adjust the capacity to reserve space.

Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

Example: The middle edge must have a flow of at least 2 and at most 4.

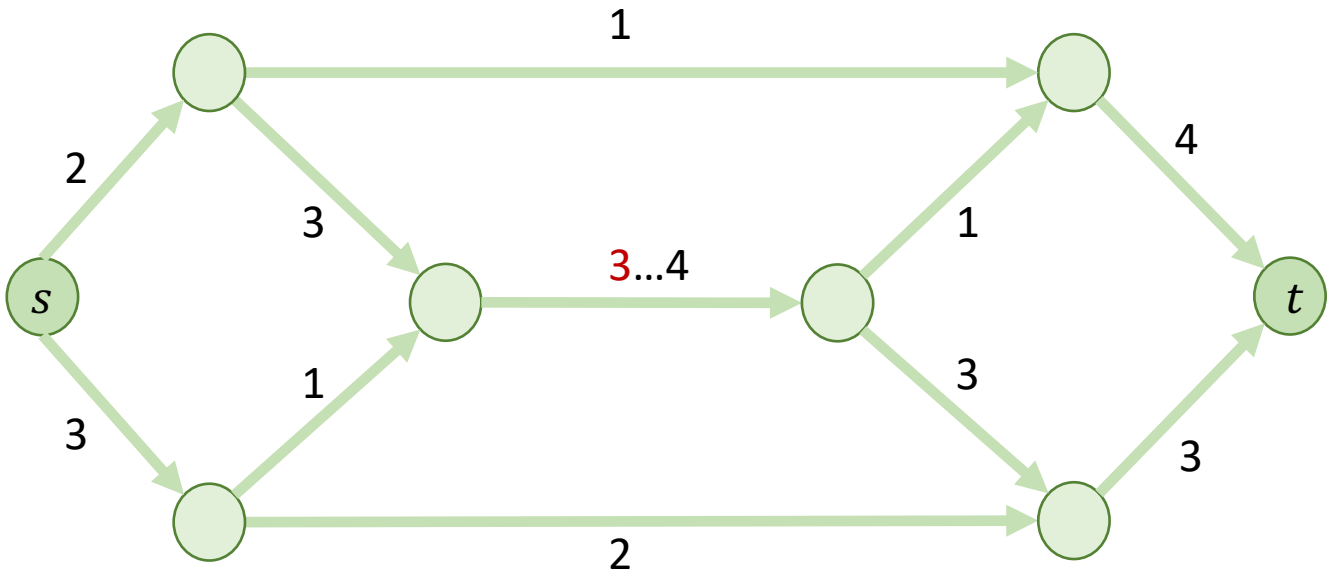


Impose a demand upstream, and produce the required amount downstream. Adjust the capacity to reserve space.

Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

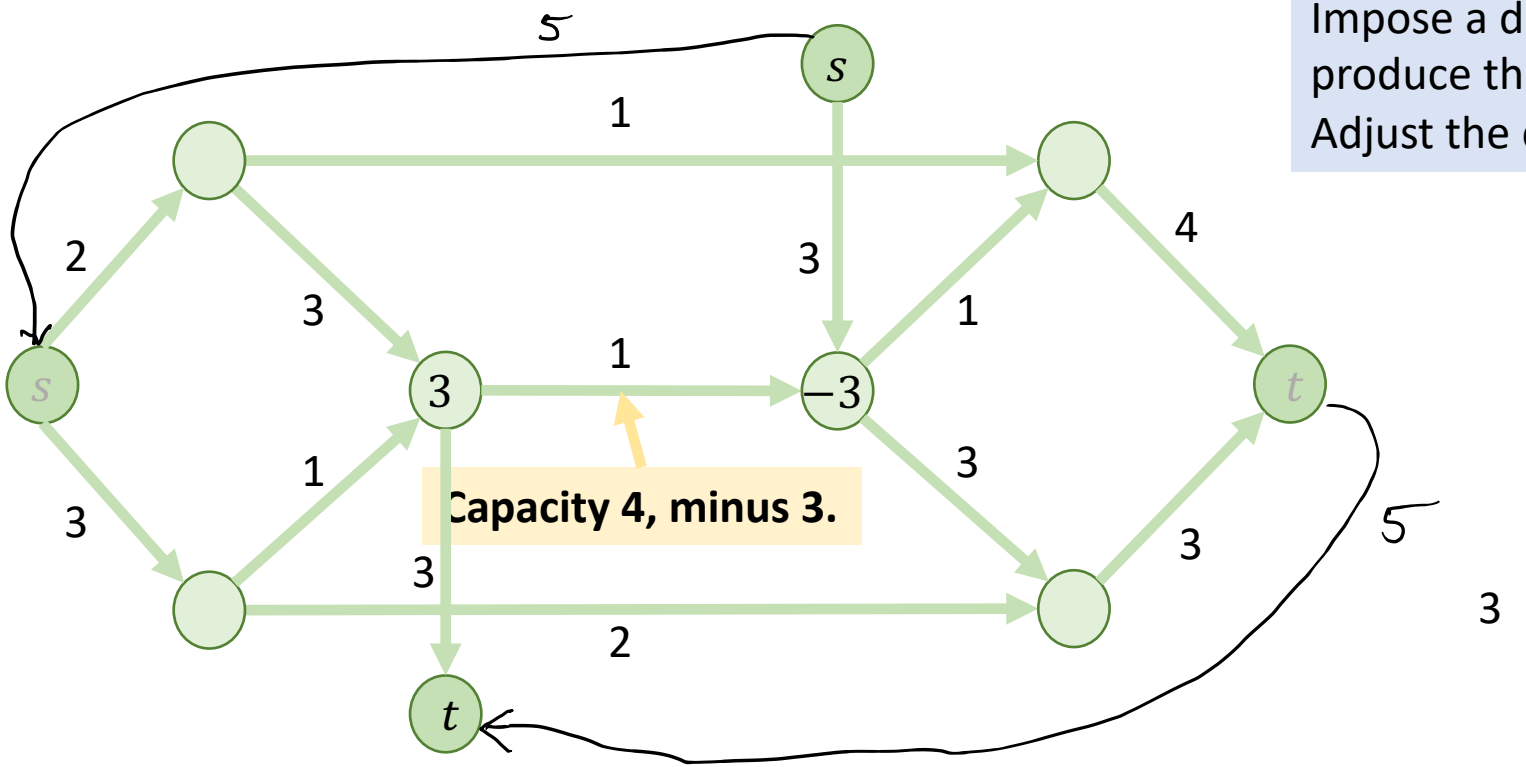
Example: The middle edge must have a flow of at least 3 and at most 4.



Minimum Capacity in Circulations

Suppose that some edges also have a minimum capacity.

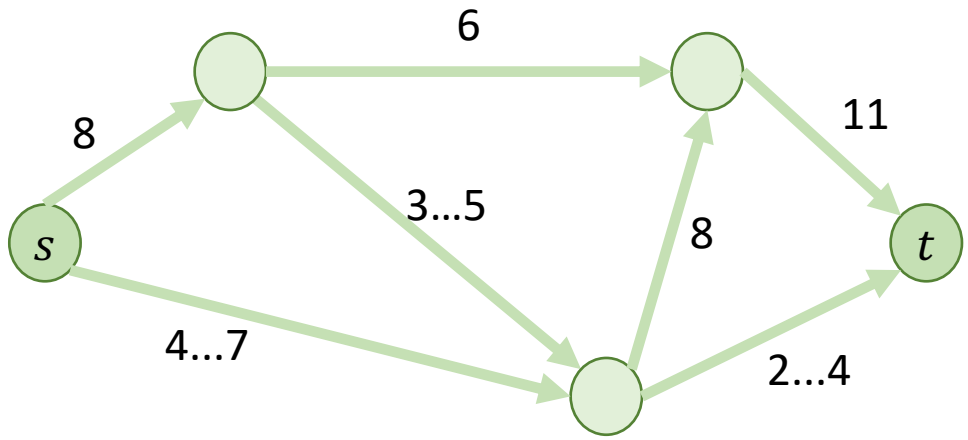
Example: The middle edge must have a flow of at least 2 and at most 4.



Impose a demand upstream, and produce the required amount downstream. Adjust the capacity to reserve space.

Min Capacity, Multiple edges

It’s likely that we see graphs with **multiple min** capacities

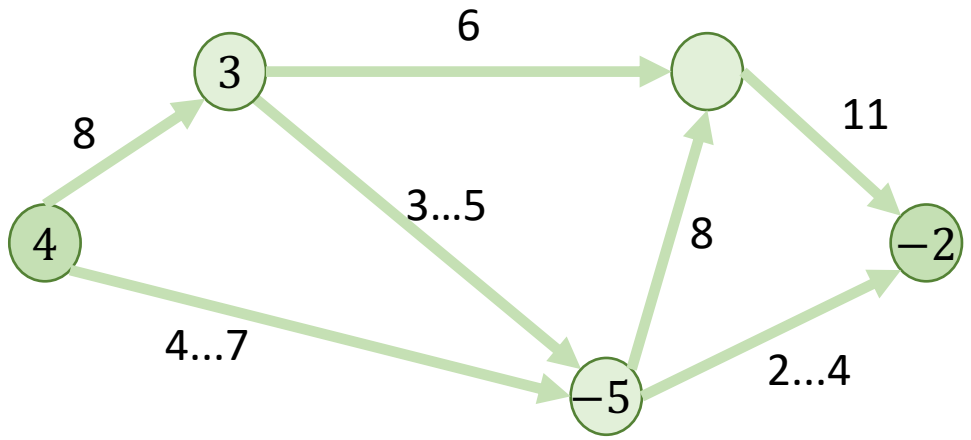


We use a similar approach.

- Fill in demands

Min Capacity, Multiple edges

It’s likely that we see graphs with **multiple min** capacities



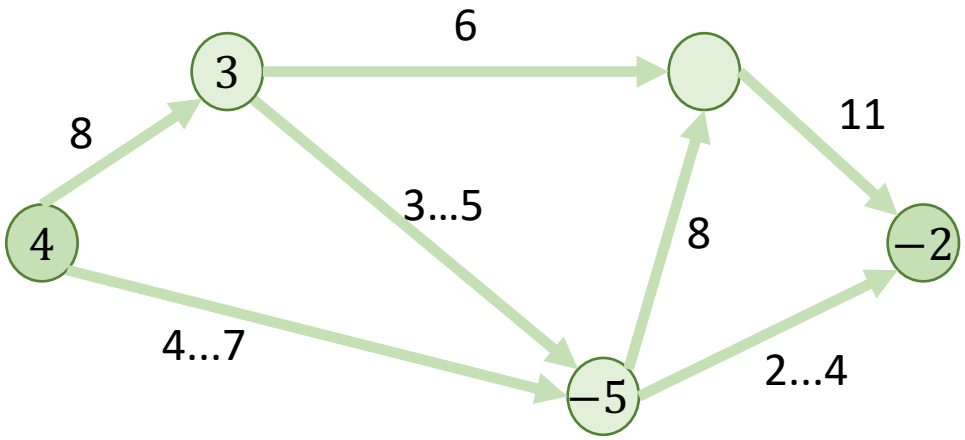
We use a similar approach.

- Fill in demands

$(-3 + -4) + (2)$ gives us -5

Min Capacity, Multiple edges

It’s likely that we see graphs with **multiple min** capacities



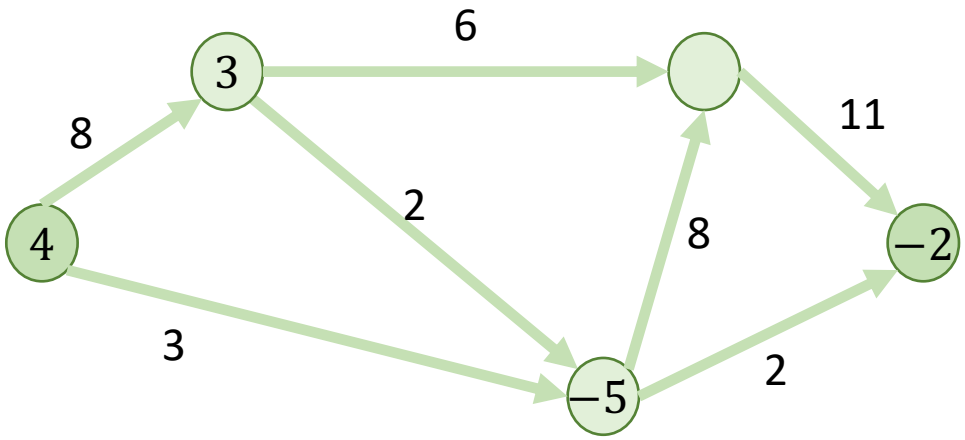
We use a similar approach.

- Fill in demands
- Adjust capacities

$(-3 + -4) + (2)$ gives us -5

Min Capacity, Multiple edges

It’s likely that we see graphs with **multiple min** capacities



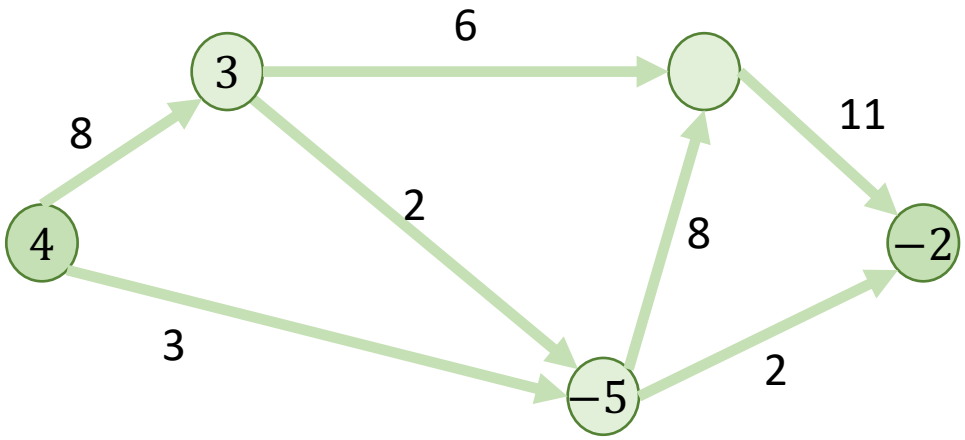
We use a similar approach.

- Fill in demands
- Adjust capacities

$(-3 + -4) + (2)$ gives us -5

Min Capacity, Multiple edges

It’s likely that we see graphs with **multiple min** capacities



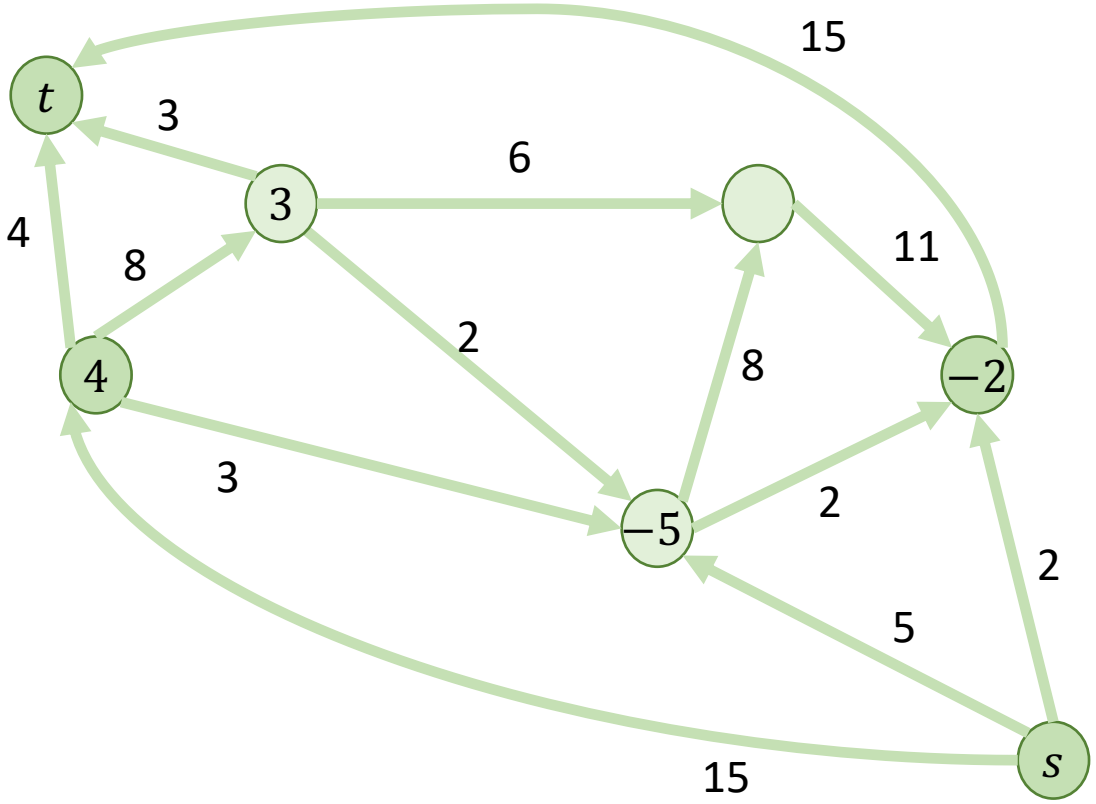
$(-3 + -4) + (2)$ gives us -5

We use a similar approach.

- Fill in demands
- Adjust capacities
- Create new sources/sinks

Min Capacity, Multiple edges

It's likely that we see graphs with **multiple min** capacities



We use a similar approach.

- Fill in demands
- Adjust capacities
- Create new sources/sinks

Flow Networks for Airline Schedules

We will now use **max flow** algorithms to plan **airline schedules**.

Given:

- A fleet of k **airplanes**.
- A set of **flight routes** (source + destination + times).

Goal:

- Find a **flight plan** which determines which planes should fly which routes.
- Every route **must be serviced** by some plane.

Flow Networks for Airline Schedules

We will now use **max flow** algorithms to plan **airline schedules**.

Given:
- A fleet of k **airplanes**.
- A set of **flight routes** (source + destination + times).

Goal:
- Find a **flight plan** which determines which planes should fly which routes.
- Every route **must be serviced** by some plane.

This is an interesting problem, because although we frame it as max flow, we are **not interested in maximizing flow** as much as we are interested in **satisfying minimums**.

Airline Schedule, Network Setup

- Our fleet has k airplanes.
- Every flight must be serviced.
- Airplanes can begin the day anywhere.
- Airplanes can end the day anywhere.
- A plane can fly its next leg out of the same airport at a later time.

Airline Schedule, Network Setup

- Our fleet has k airplanes.

  The source has a **capacity of k**.

- Every flight must be serviced.

- Airplanes can begin the day anywhere.

- Airplanes can end the day anywhere.

- A plane can fly its next leg out of the same airport at a later time.

Airline Schedule, Network Setup

- Our fleet has k airplanes.

  The source has a **capacity of k**.

- Every flight must be serviced.

  A flight is represented by an edge with **both min and max capacity = 1**.

- Airplanes can begin the day anywhere.

- Airplanes can end the day anywhere.

- A plane can fly its next leg out of the same airport at a later time.

Airline Schedule, Network Setup

- Our fleet has k airplanes.

  The source has a **capacity of k**.

- Every flight must be serviced.

  A flight is represented by an edge with **both min and max capacity = 1**.

- Airplanes can begin the day anywhere.

  Edge from the source to the start of **every flight**.

- Airplanes can end the day anywhere.

- A plane can fly its next leg out of the same airport at a later time.

Airline Schedule, Network Setup

- Our fleet has k airplanes.

  The source has a **capacity of k**.

- Every flight must be serviced.

  A flight is represented by an edge with **both min and max capacity = 1**.

- Airplanes can begin the day anywhere.

  Edge from the source to the start of **every flight**.

- Airplanes can end the day anywhere.

  Edge from the end of **every flight** to the sink

- A plane can fly its next leg out of the same airport at a later time.

Airline Schedule, Network Setup

- Our fleet has k airplanes.

  The source has a **capacity of k**.

- Every flight must be serviced.

  A flight is represented by an edge with **both min and max capacity = 1**.

- Airplanes can begin the day anywhere.

  Edge from the source to the start of **every flight**.

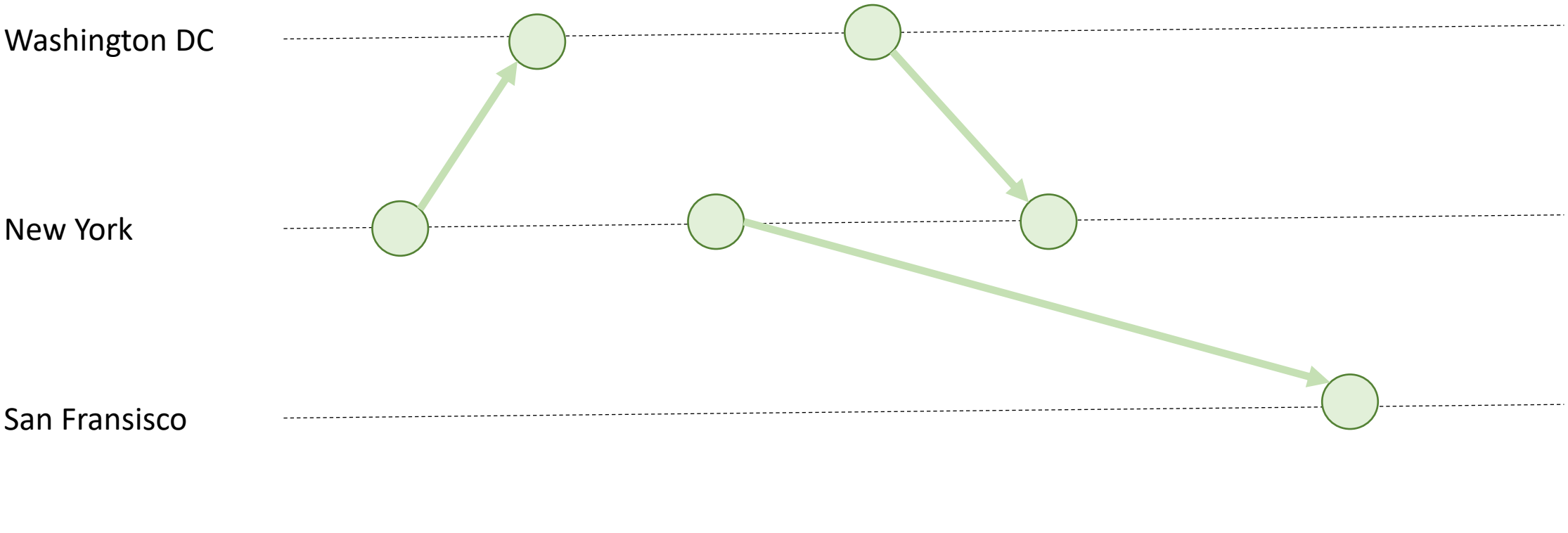
- Airplanes can end the day anywhere.

  Edge from the end of **every flight** to the sink

- A plane can fly its next leg out of the same airport at a later time.

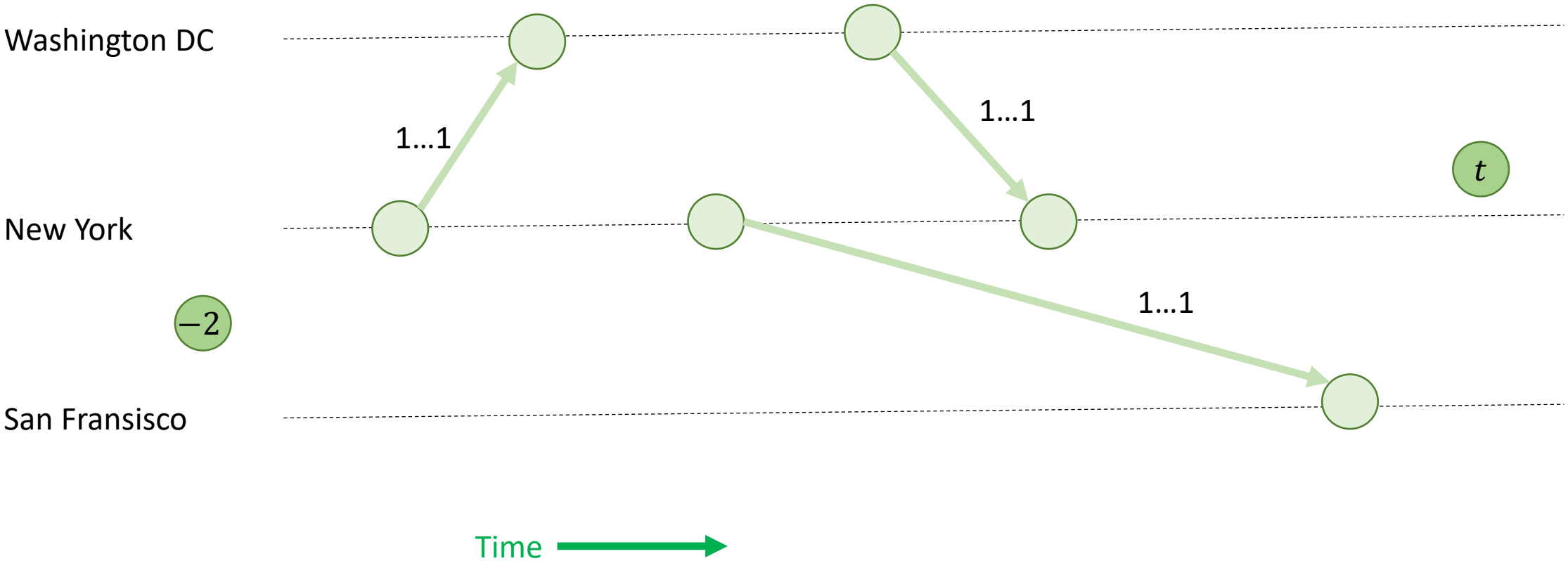
  Capacity 1 edges from the **end of any flight** to the **start of all other flights** at the same airport + later time.

Airline Schedule, Example

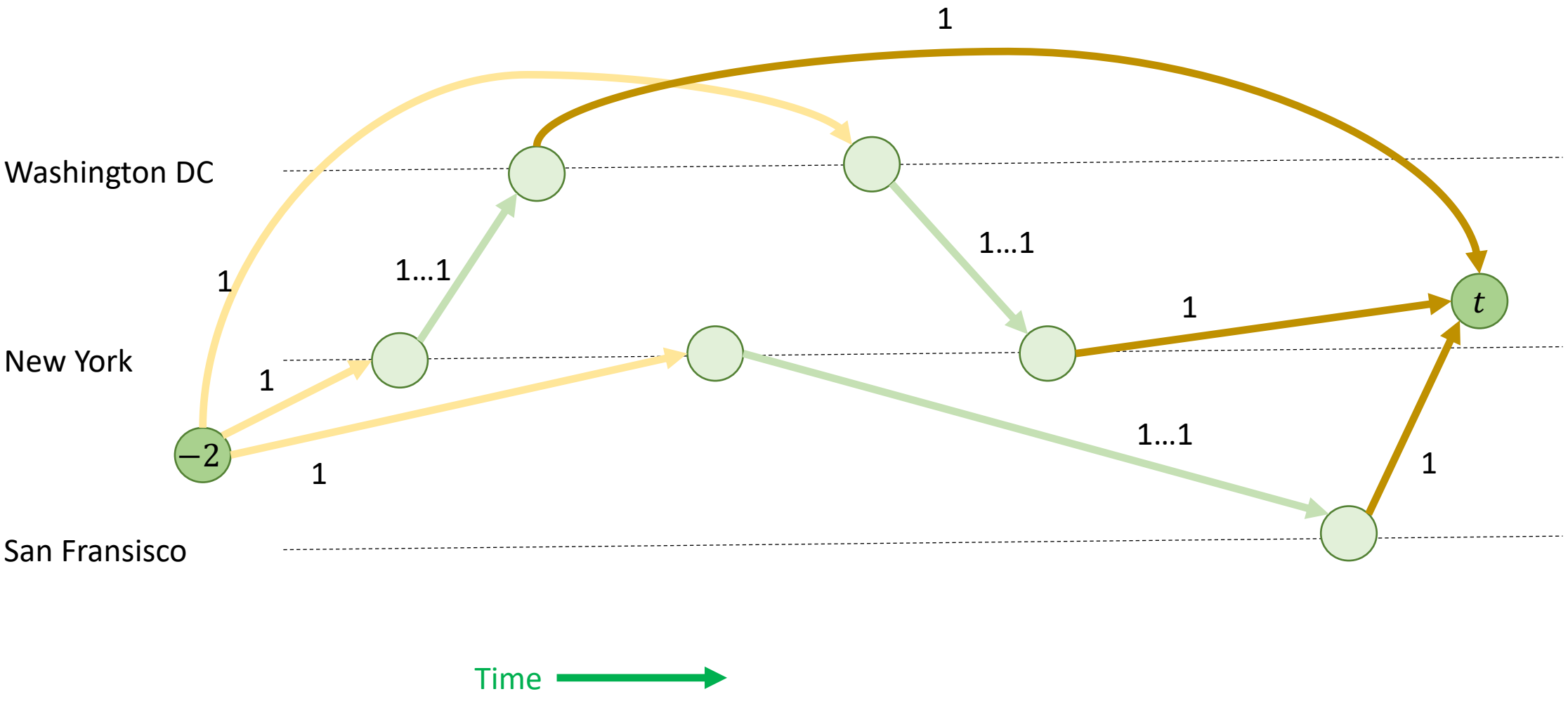


Assume $k = 2$ airplanes.

Airline Schedule, Example



Airline Schedule, Example



Airline Schedule, Example