Lecture 16

Max Flows, Min Cuts, and Ford-Fulkerson

Announcements

- HW7 Friday!
  - The last one!

Announcements: Final exam!

- Final Exam will be **December 10 at 8:30am**
  - Logistics coming soon
- The exam is technically cumulative, but focusing on Lectures 10-16 (today).

- Sample exams published already (see website)
- “Official” practice exam coming soon

Aside:

Public Service Announcement

- How to best prepare for the final?

  not having a concussion during

- ***Bike Safety!***
  - Wear a helmet!
  - Put your phone away while biking!
  - Use rotaries as intended!

YIELD
YIELD
YIELD

End of announcements!

The plan for today

- Minimum s-t cuts
- Maximum s-t flows
- The Ford-Fulkerson Algorithm
  - Finds min cuts and max flows!
- Applications
  - Why do we want to find these things?

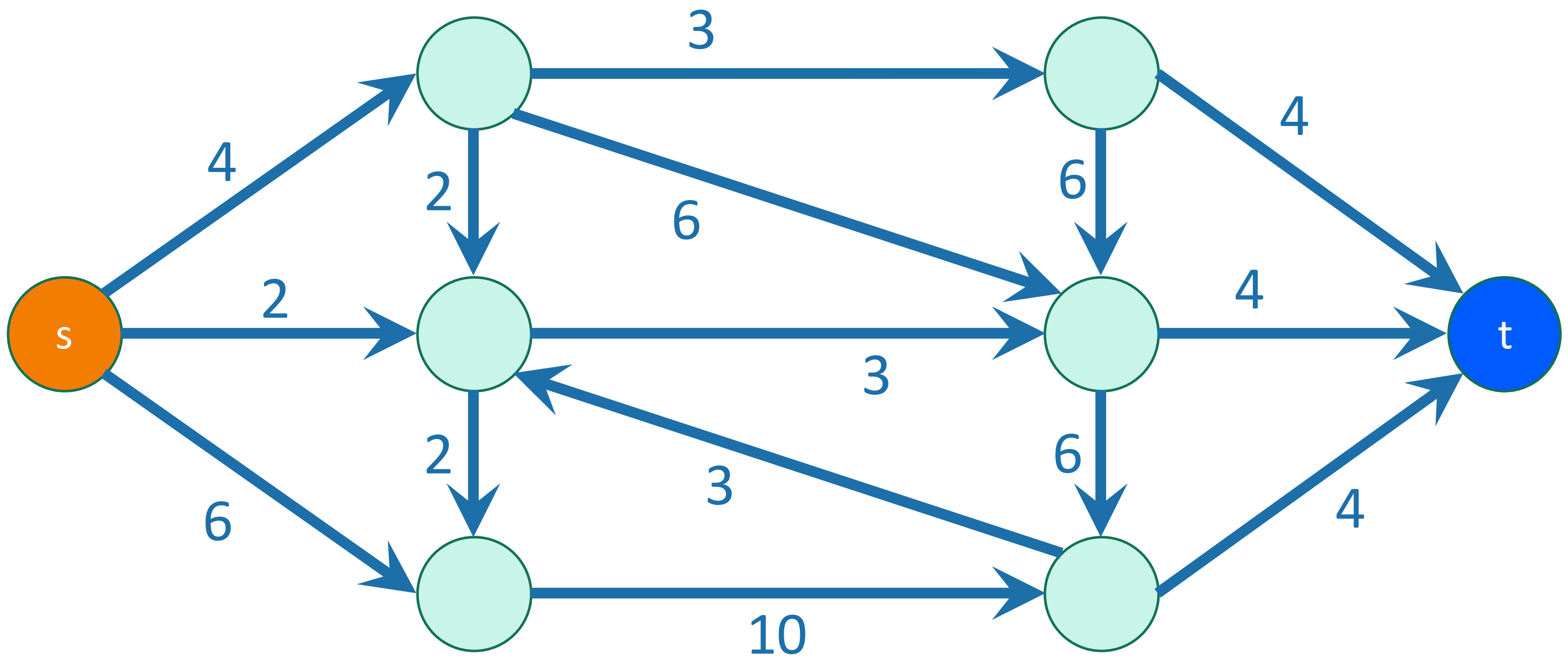
This lecture will skip a few proofs, and you are **not** responsible for the proofs for the final exam.

Lucky the lackadaisical lemur

Min cuts and max flows

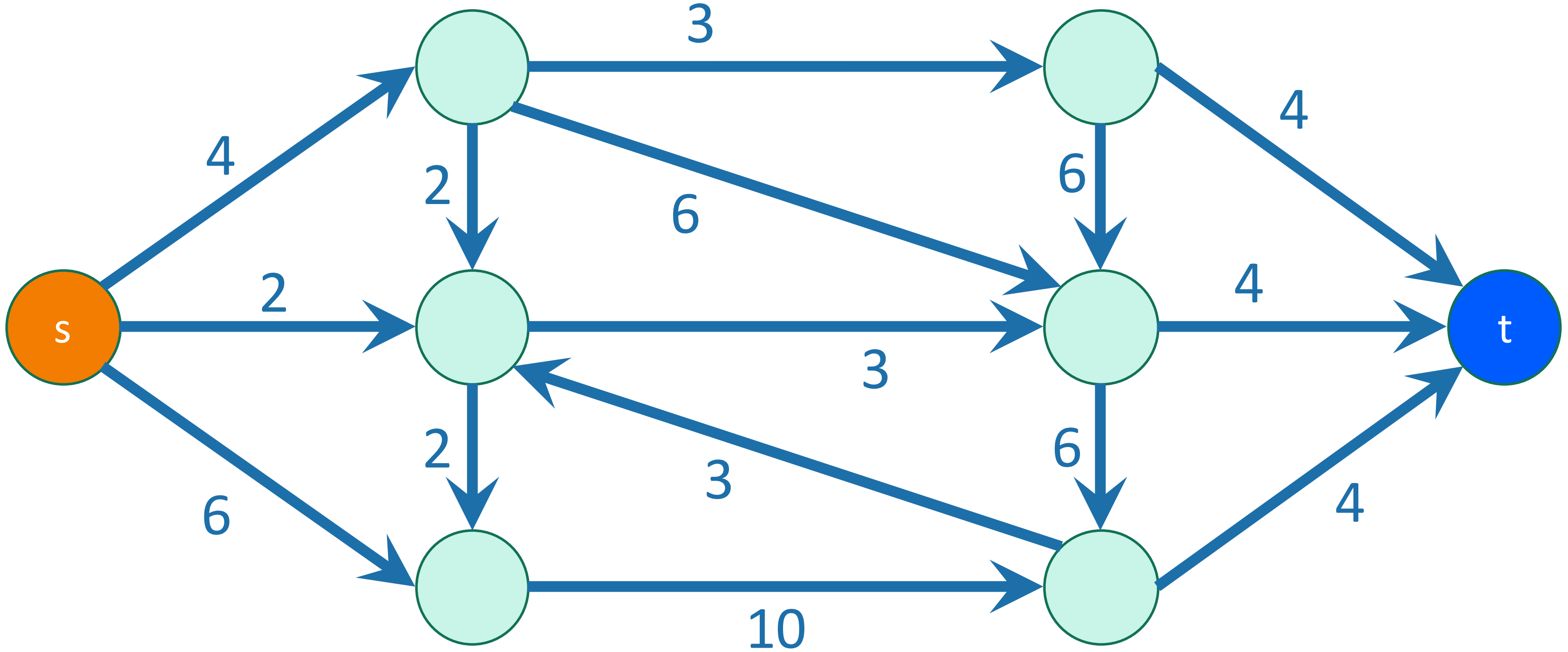
Set-up for today

- Graphs are directed and edges have "capacities" (weights)
- We have a special "source" vertex s and "sink" vertex t.
    - s has only outgoing edges
    - t has only incoming edges



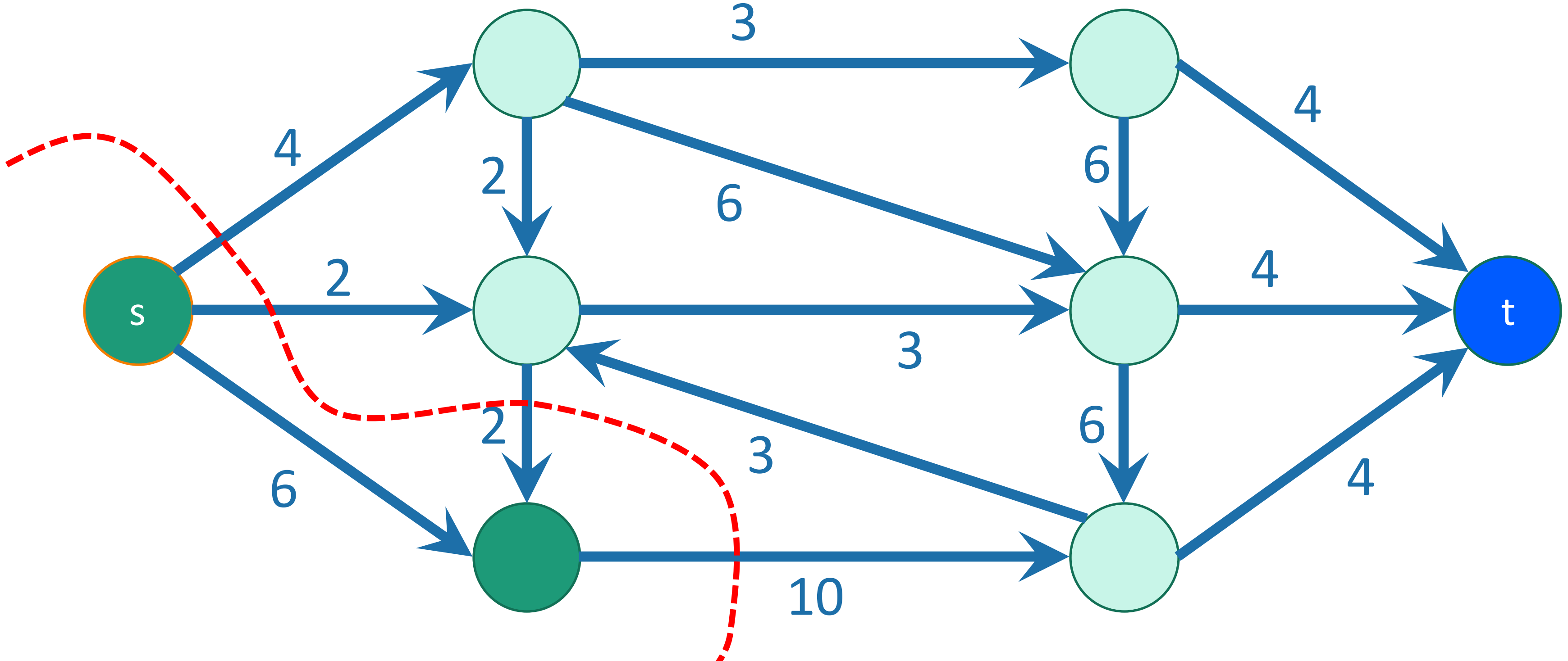
An **s-t cut**

is a cut which separates s from t



An **s-t cut**

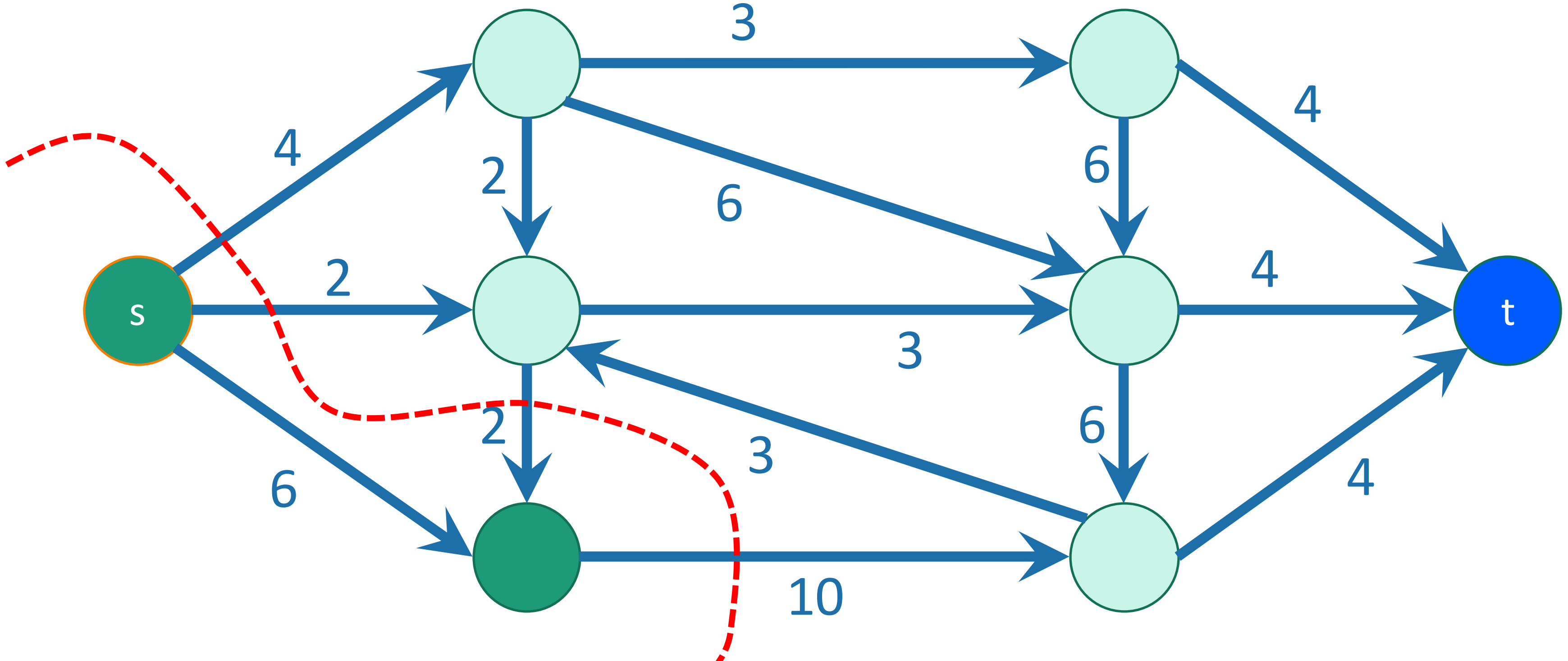
is a cut which separates s from t



An **s-t cut**

is a cut which separates s from t

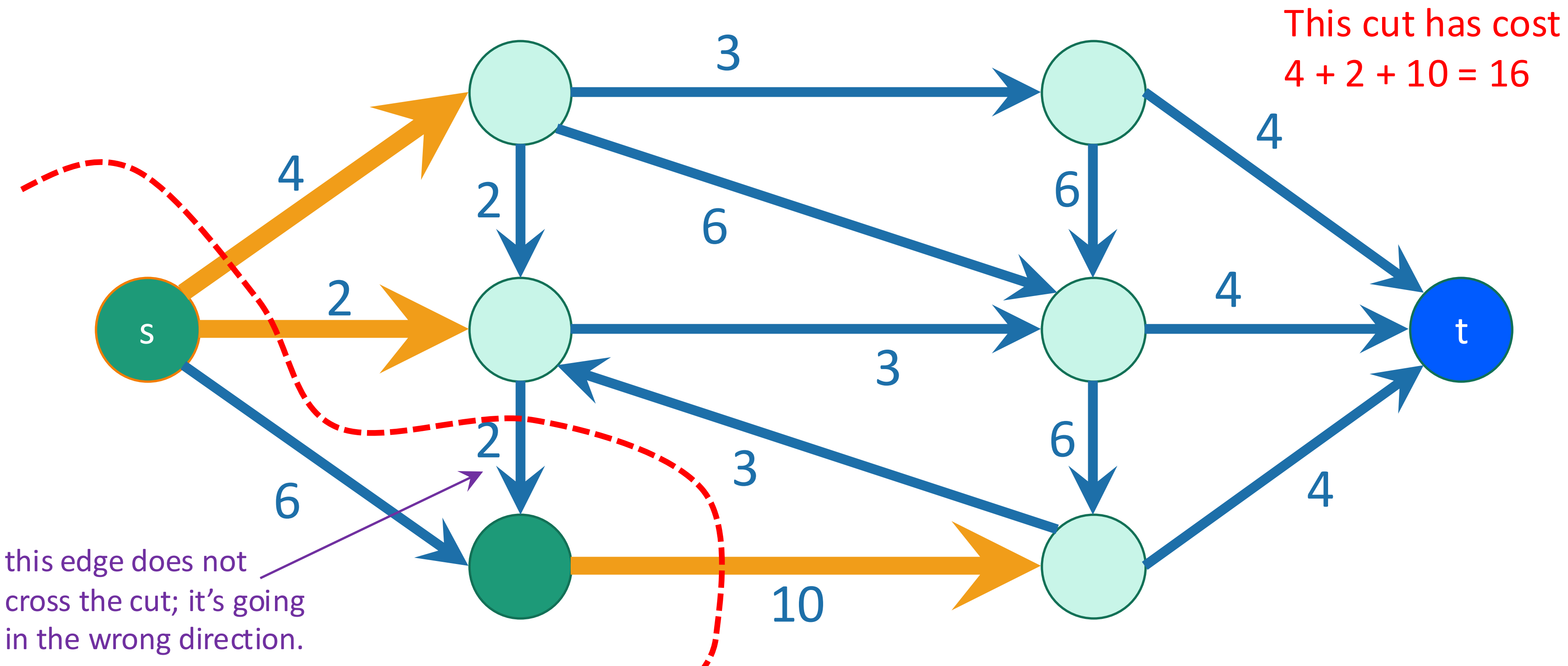
- An edge **crosses the cut** if it goes from s's side to t's side.



An **s-t cut**

is a cut which separates s from t

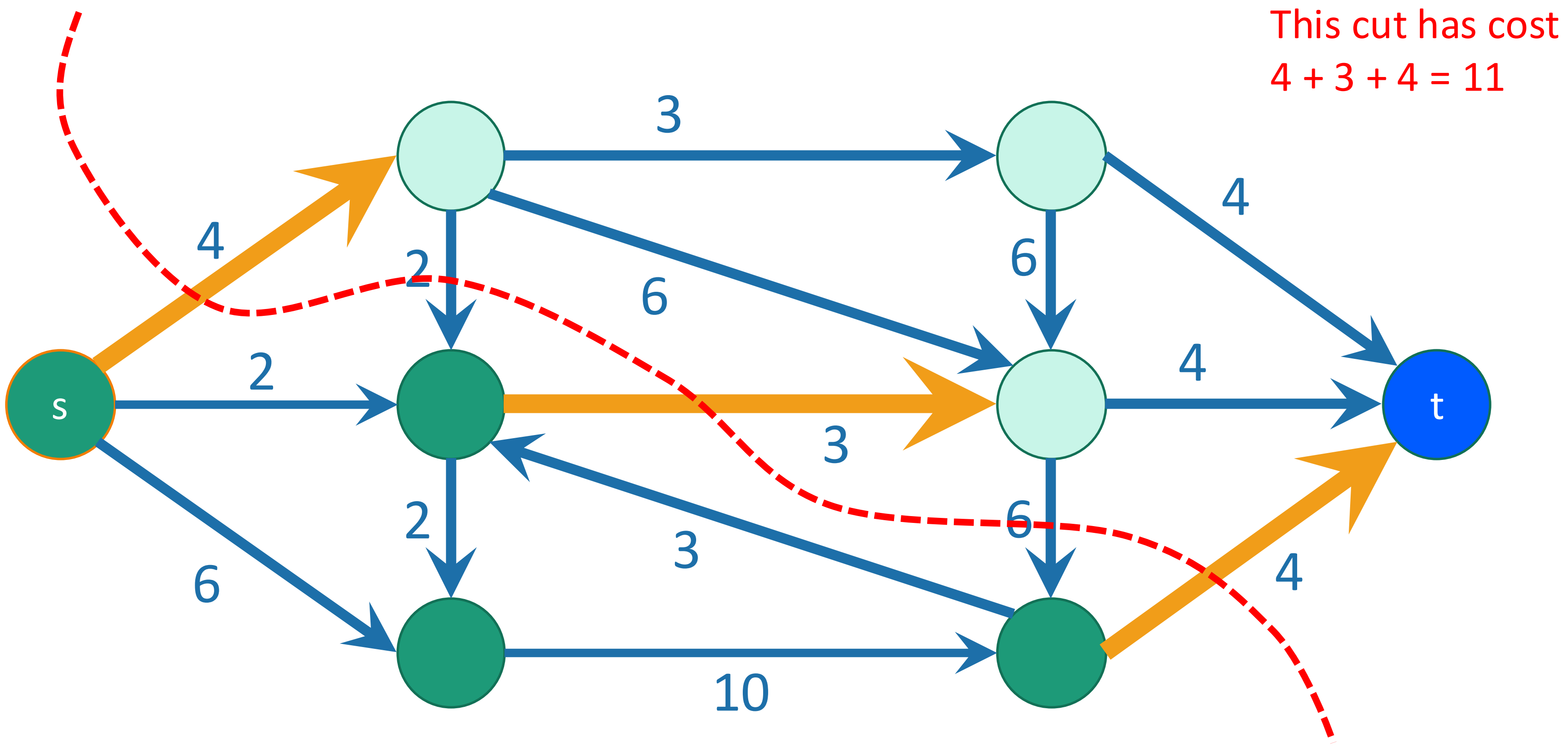
- An edge **crosses the cut** if it goes from s's side to t's side.
- The **cost** (or capacity) of a cut is the sum of the capacities of the edges that cross the cut.


This cut has cost
4 + 2 + 10 = 16
s
t
3
4
2
6
6
4
2
3
4
2
6
3
6
4
10
this edge does not cross the cut; it's going in the wrong direction.

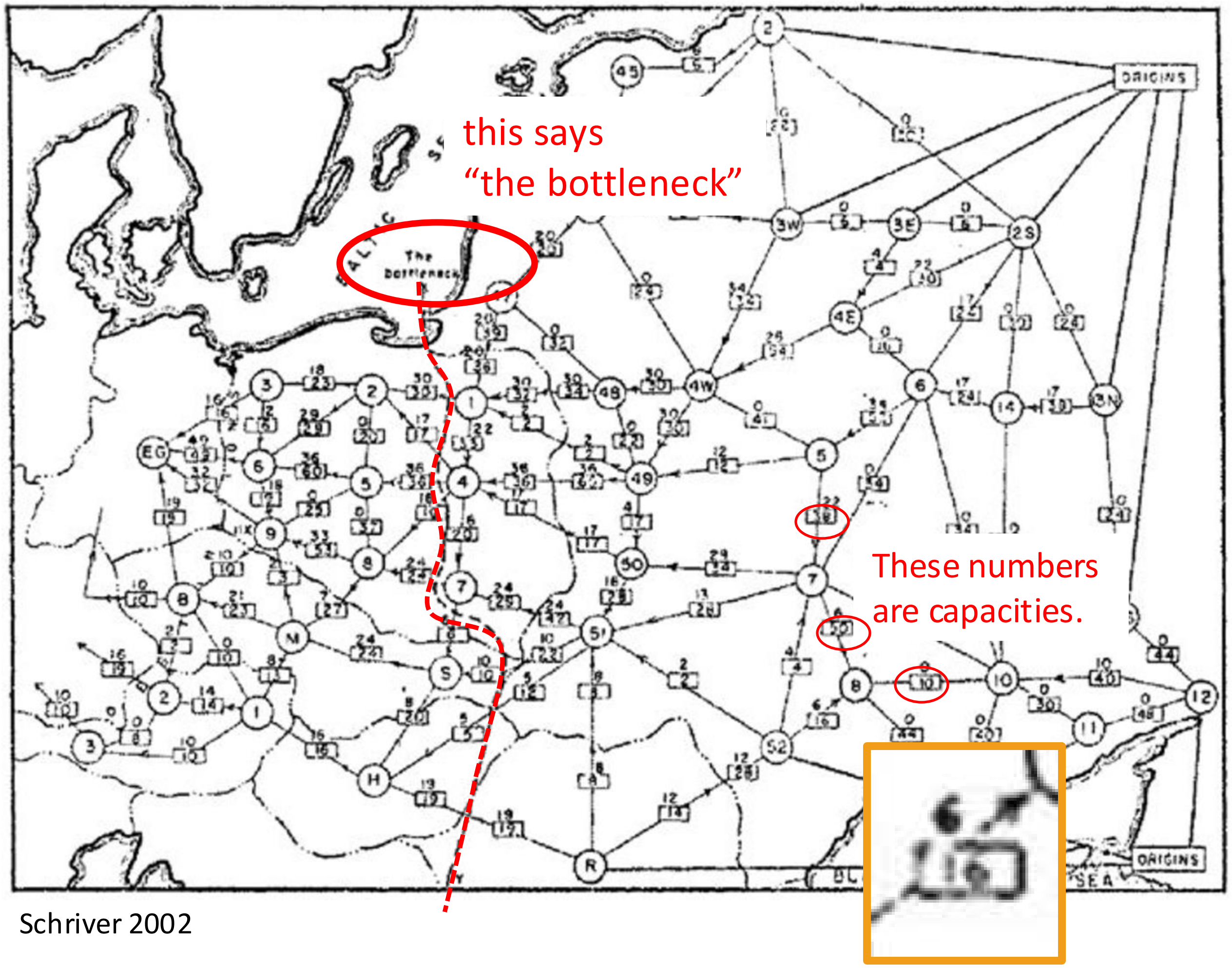
A minimum **s-t cut**

is a cut which separates s from t
with minimum cost.

- Question: how do we find a minimum s-t cut?



Example where this comes up

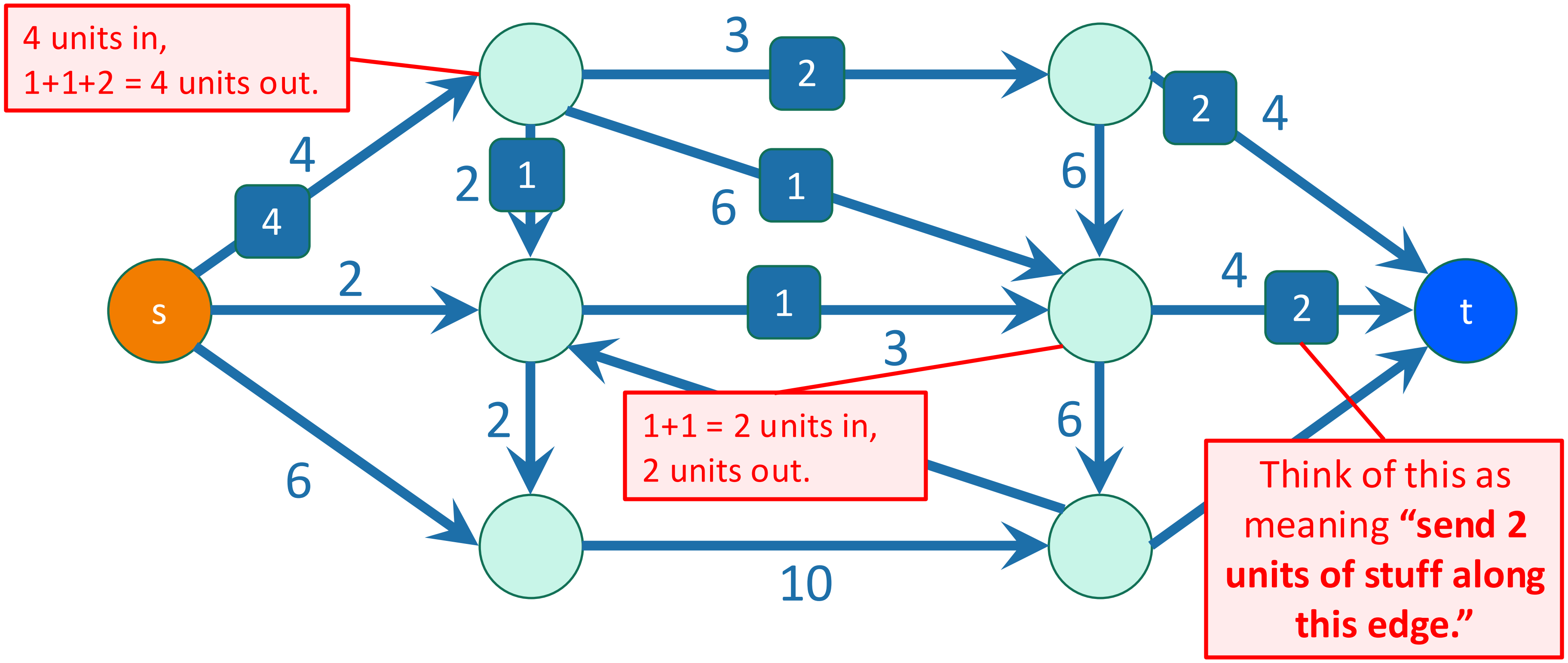


Schriver 2002

- 1955 map of rail networks from the Soviet Union to Eastern Europe.
    - Declassified in 1999.
    - 44 edges, 105 vertices
- The US wanted to cut off routes from suppliers in Russia to Eastern Europe as efficiently as possible.
- In 1955, Ford and Fulkerson gave an algorithm which finds the optimal s-t cut.

Flows

- In addition to a capacity, each edge has a flow
  - (unmarked edges in the picture have flow 0)
- The flow on an edge must be at most its capacity.
- At each vertex, the incoming flows must equal the outgoing flows.

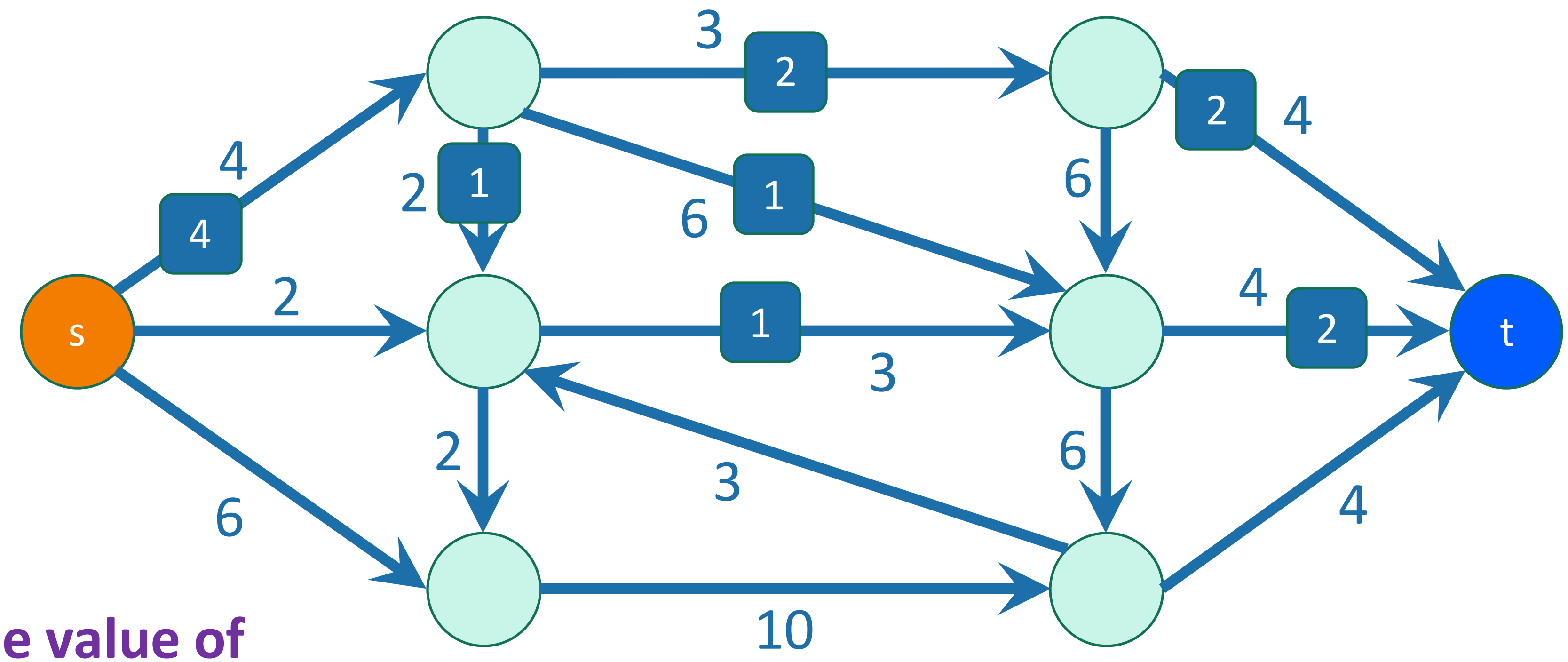


Flows

- The value of a flow is:
  - The amount of stuff coming out of s
  - The amount of stuff flowing into t
  - These are the same!

Because of conservation of flows at vertices,

stuff you put in
=
stuff you take out.

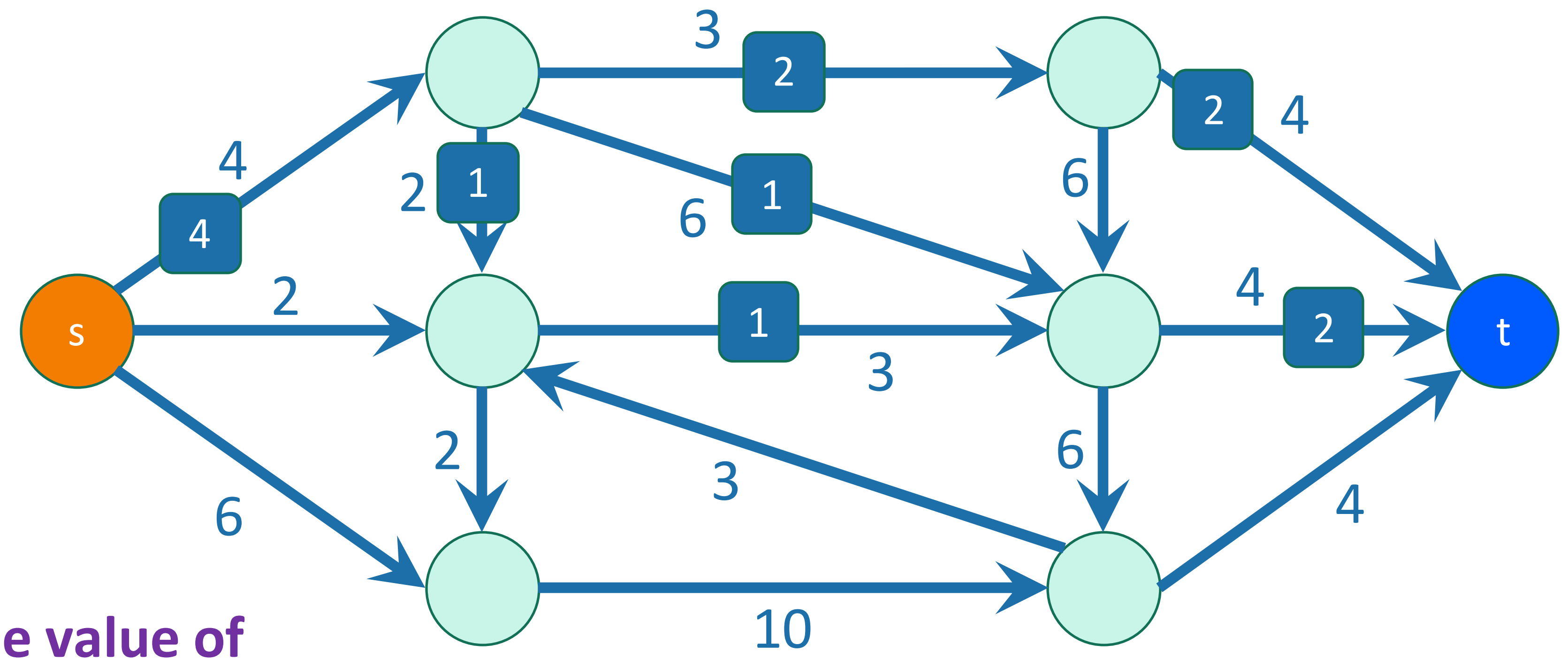


The value of this flow is 4.

A maximum flow

is a flow of maximum value.

- This example flow is pretty wasteful, I'm not utilizing the capacities very well.

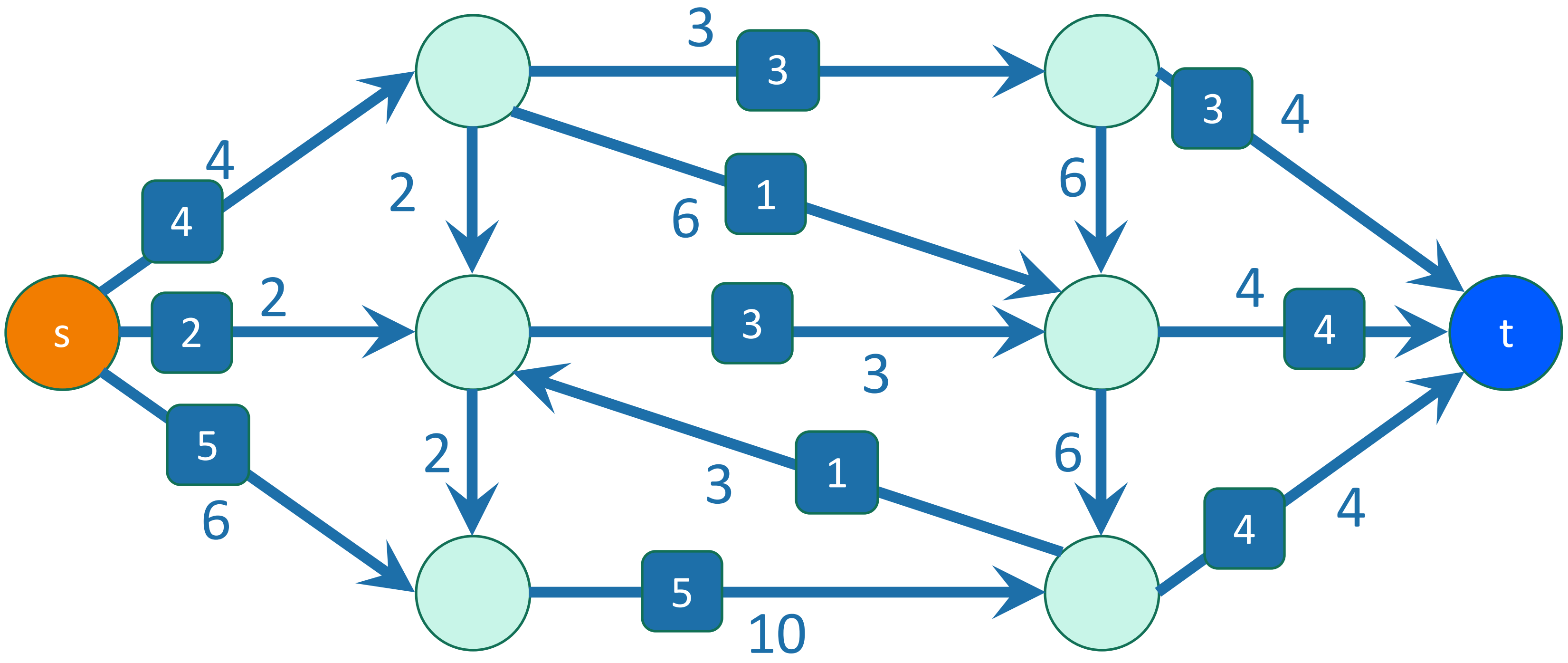


The value of this flow is 4.

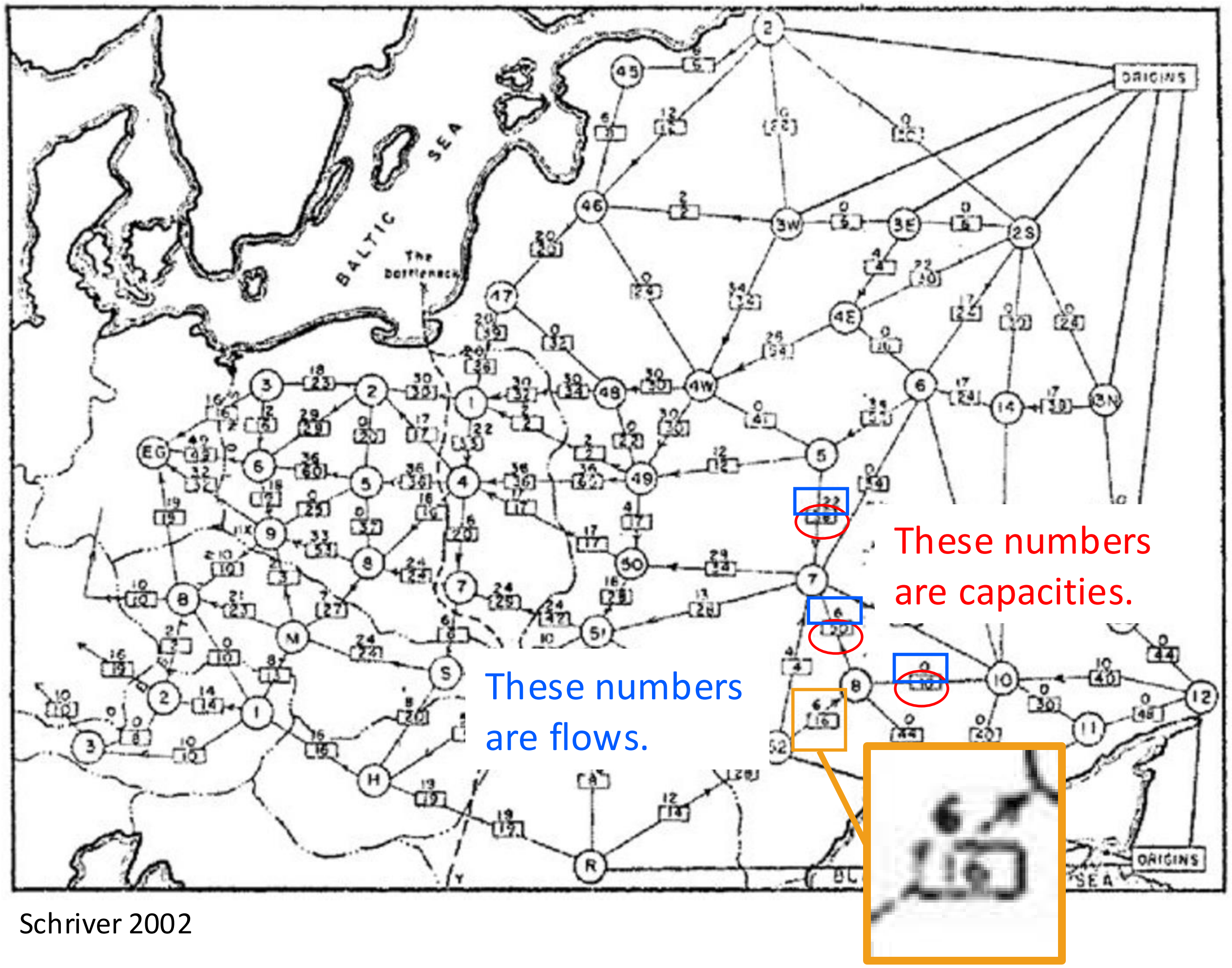
A maximum flow

is a flow of maximum value.

- This one is maximum; it has value 11.



Example where this comes up



Schriver 2002

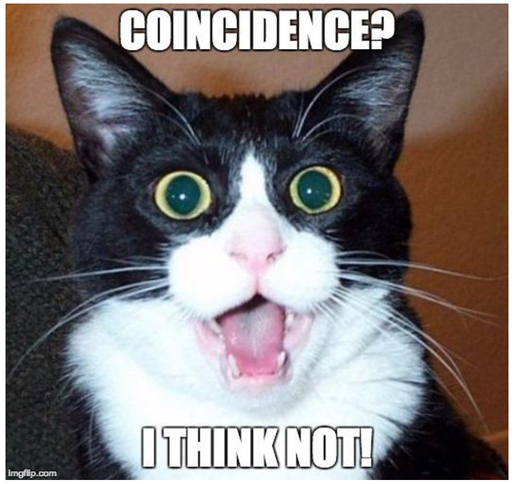
- 1955 map of rail networks from the Soviet Union to Eastern Europe.
    - Declassified in 1999.
    - 44 edges, 105 vertices
- The Soviet Union wants to route supplies from suppliers in Russia to Eastern Europe as efficiently as possible.

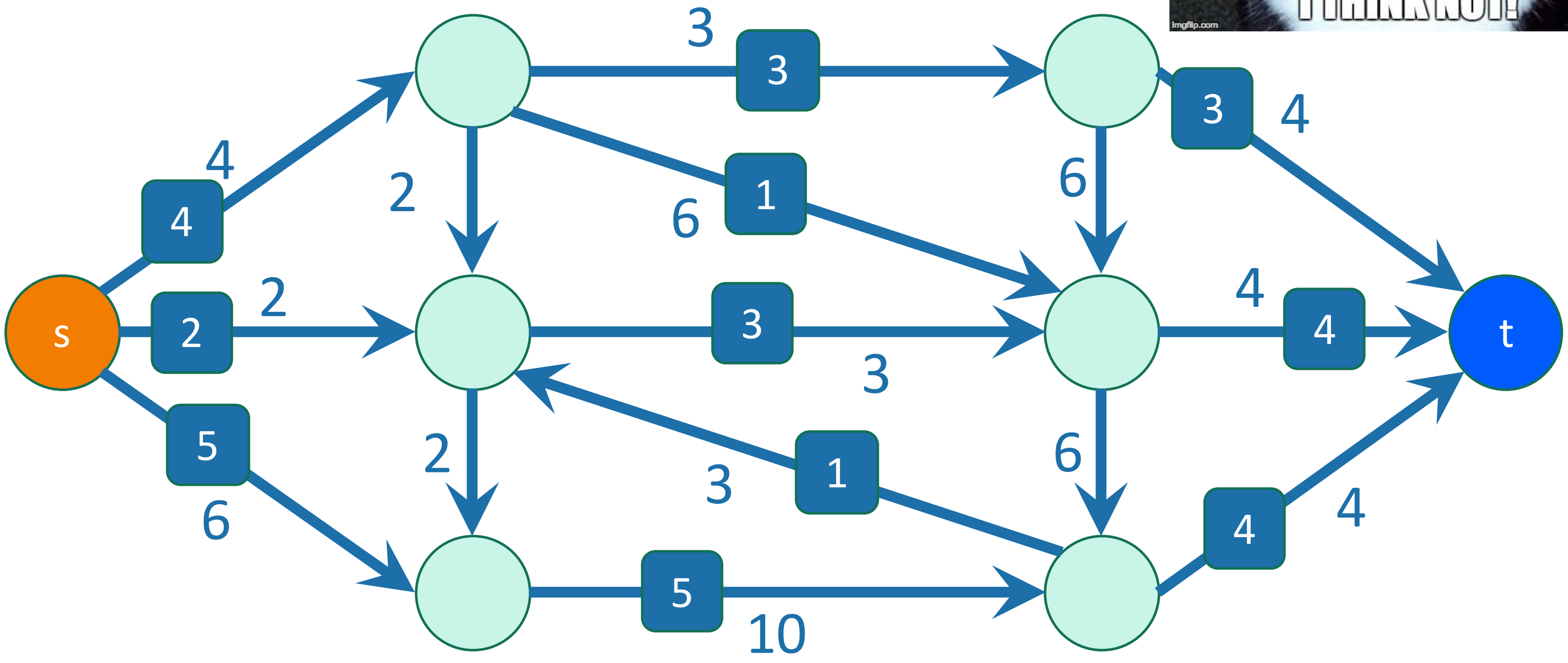
A maximum flow

is a flow of maximum value.

- This one is maximal; it has value 11.

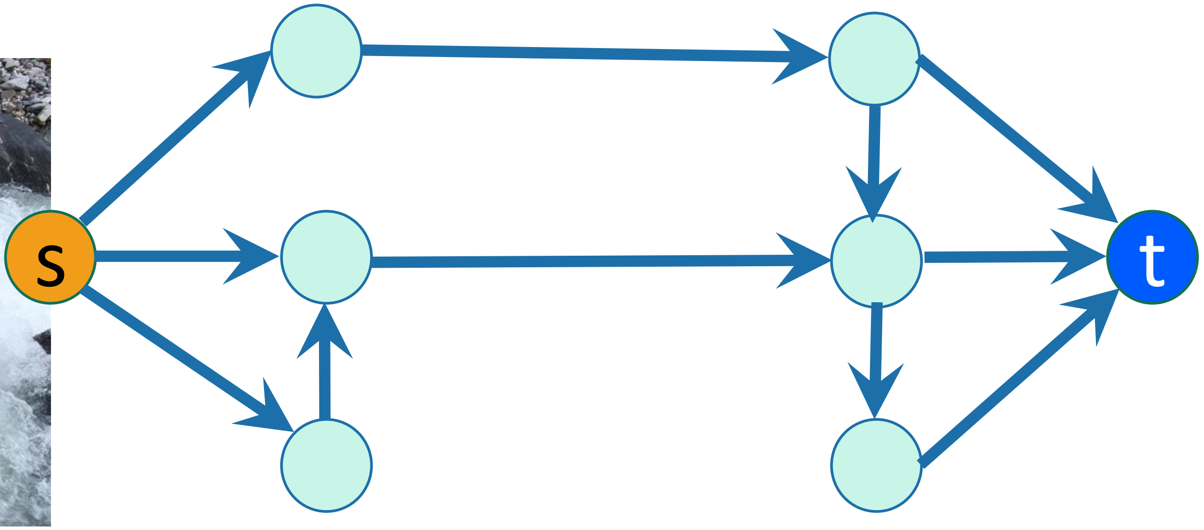
That's the same as the minimum cut in this graph!





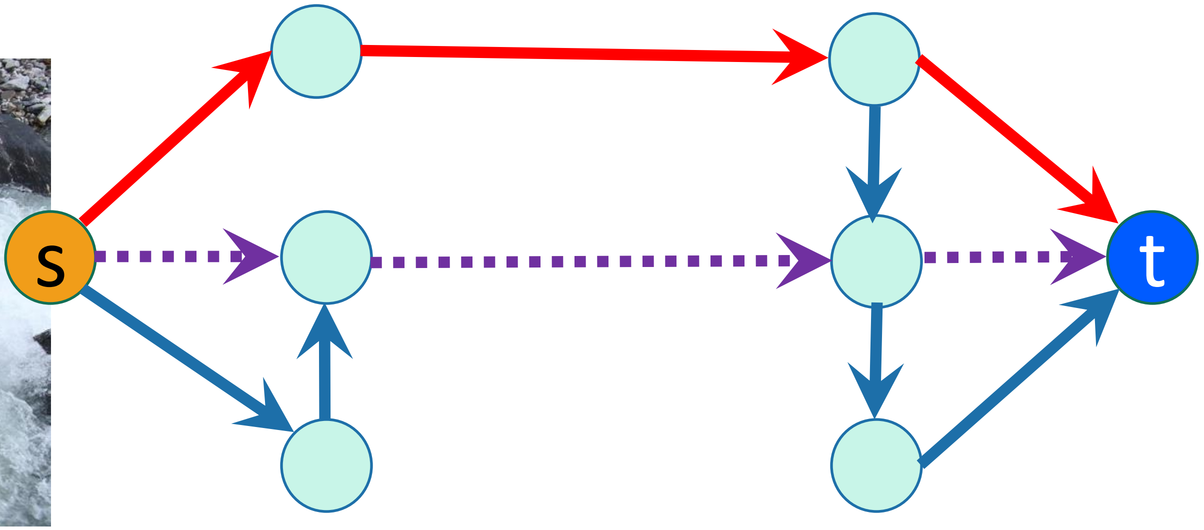
Pre-lecture exercise

- Each edge is a (directed) rickety bridge.
- How many bridges need to fall down to disconnect s from t? For this graph, 2
- If only one person can be on a bridge at a time, and you want to keep traffic moving (aka, no waiting at vertices allowed), how many people can get to t at a time? Also 2!



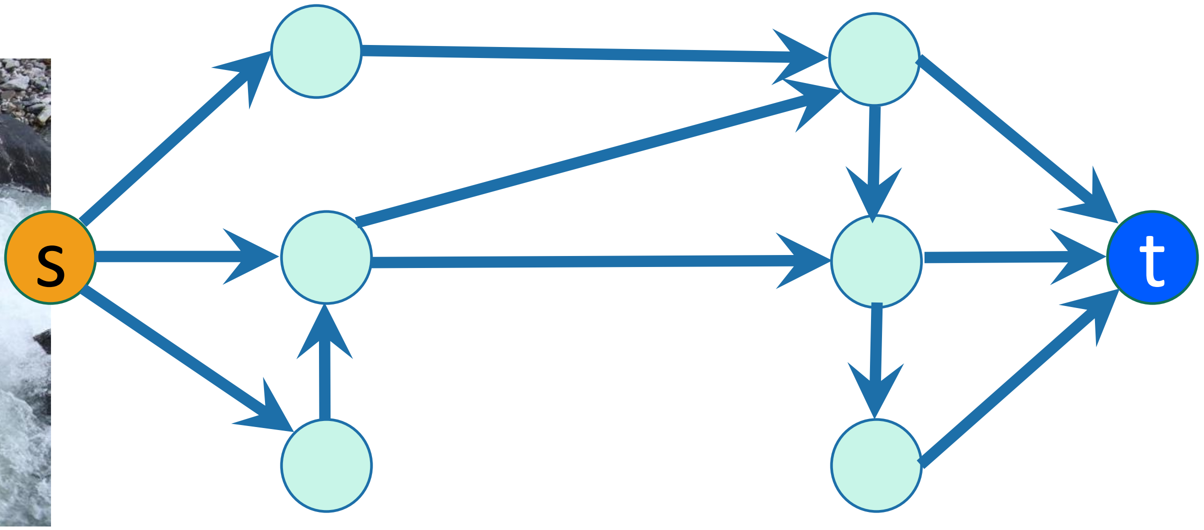
Pre-lecture exercise

- Each edge is a (directed) rickety bridge.
- How many bridges need to fall down to disconnect s from t? For this graph, 2
- If only one person can be on a bridge at a time, and you want to keep traffic moving (aka, no waiting at vertices allowed), how many people can get to t at a time? Also 2!



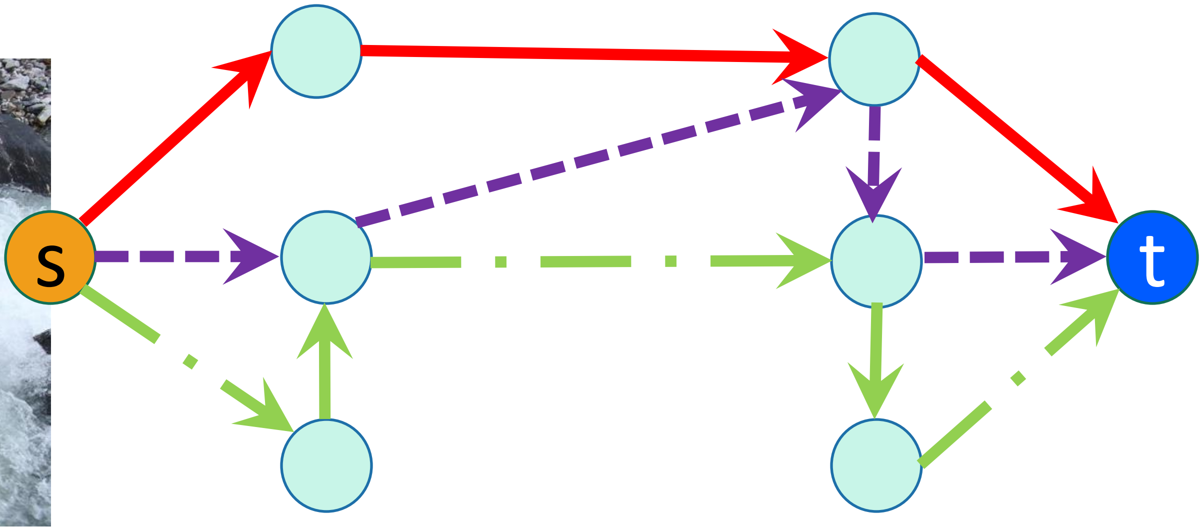
How about now?

- Each edge is a (directed) rickety bridge.
- How many bridges need to fall down to disconnect s from t? For this graph, 3
- If only one person can be on a bridge at a time, and you want to keep traffic moving (aka, no waiting at vertices allowed), how many people can get to t at a time? Also 3!



How about now?

- Each edge is a (directed) rickety bridge.
- How many bridges need to fall down to disconnect s from t? For this graph, 3
- If only one person can be on a bridge at a time, and you want to keep traffic moving (aka, no waiting at vertices allowed), how many people can get to t at a time? Also 3!



Pre-lecture exercise

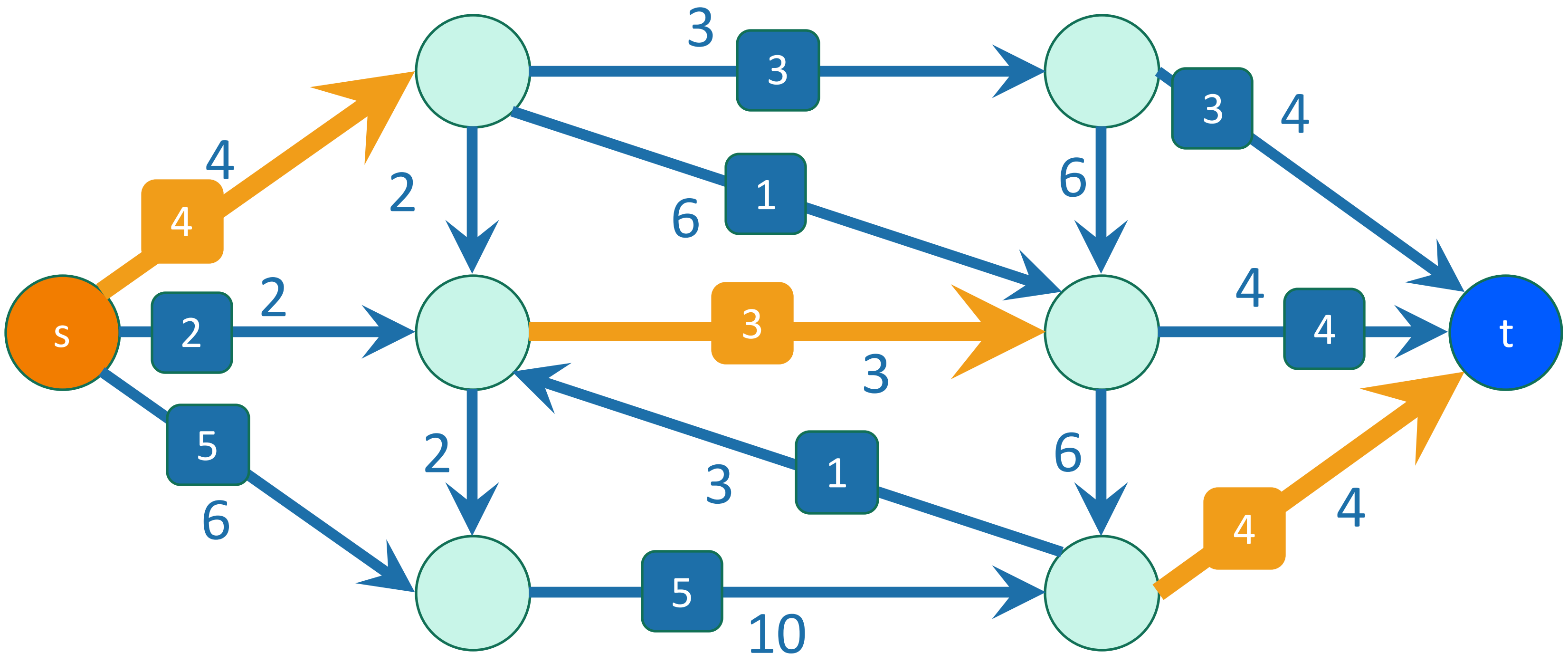
- Can you come up with a graph where the two numbers are different?

Theorem

Max-flow min-cut theorem

The value of a max flow from s to t
is equal to
the cost of a min s-t cut.

Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.

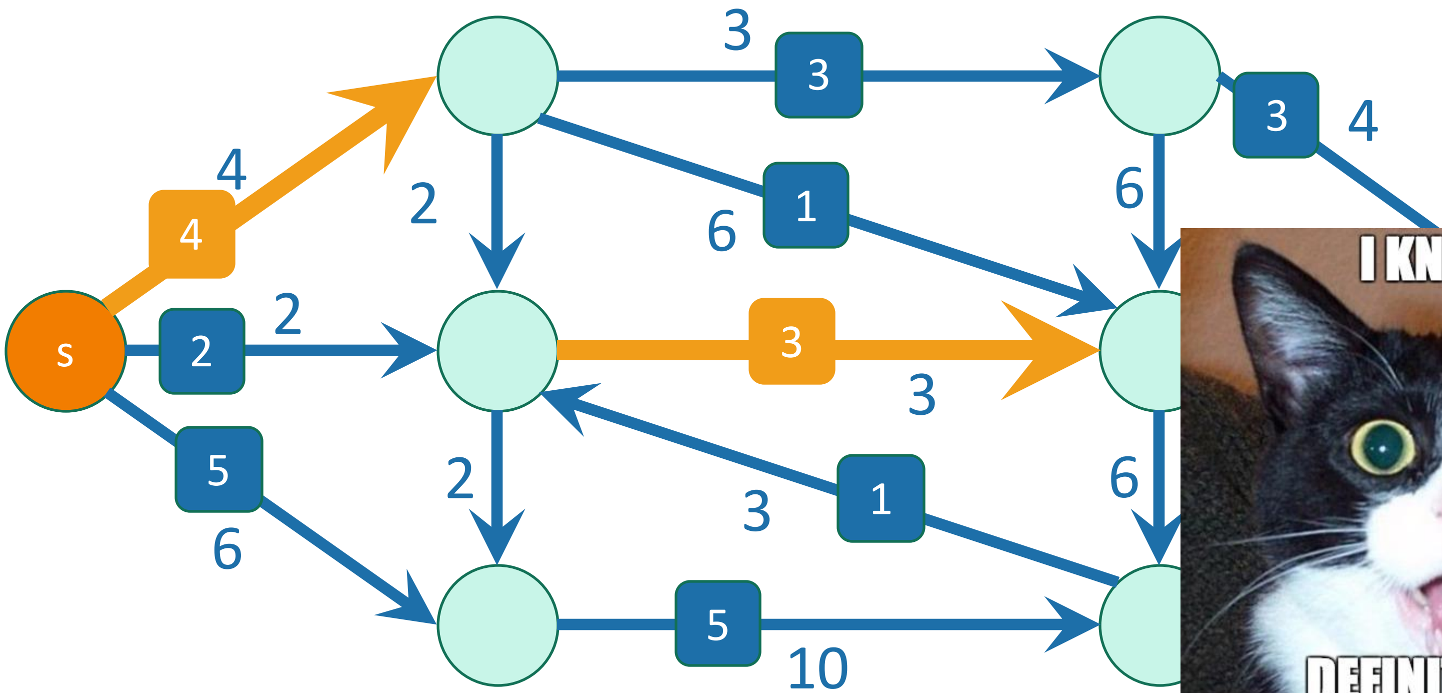


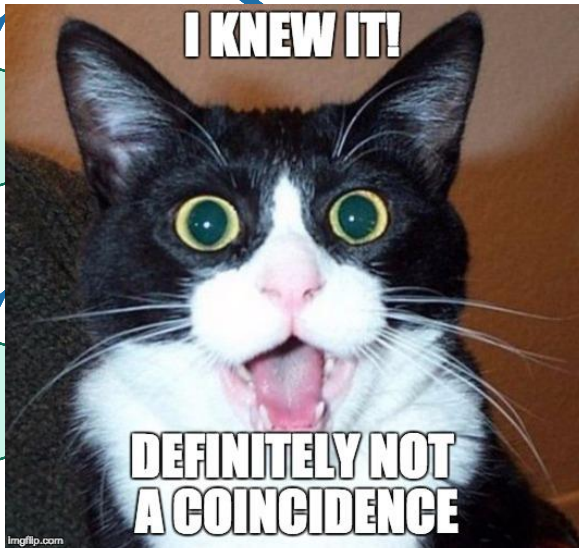
Theorem

Max-flow min-cut theorem

The value of a max flow from s to t
is equal to
the cost of a min s-t cut.

Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.





Useful corollary

- Suppose that you can find:
  - An s-t cut of cost X
  - An s-t flow with value X

- Then the minimum s-t cut and the maximum s-t flow must *both* be equal to X.

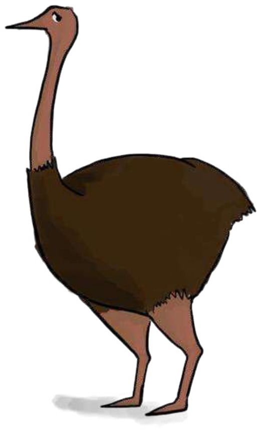
$$X \geq \text{Min cut cost} = \text{Max flow value} \geq X$$

$\Rightarrow$ All of these things must be equal!

We will skip the proof
of Min-Cut Max-Flow Theorem

- You don’t need to know the proof of the Min-Cut Max-Flow theorem for this course.
- Instead, we will focus on how to find max flows and min cuts, and applications.

Check out the proof in CLRS if you are curious!

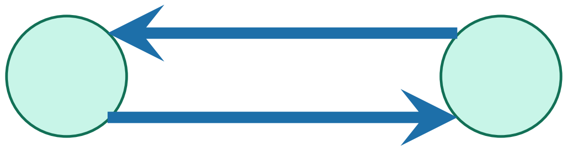


Ford-Fulkerson Algorithm

Ford-Fulkerson algorithm

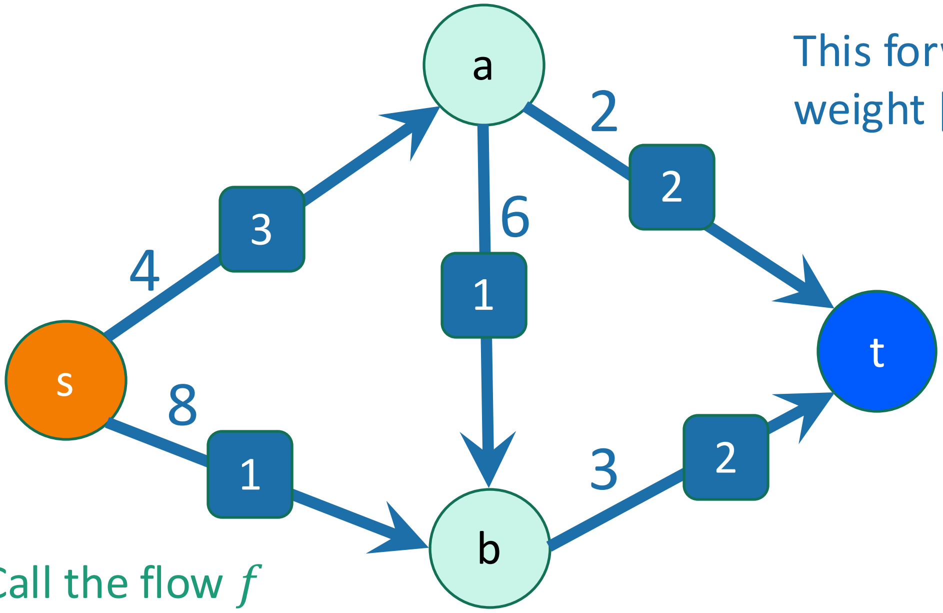
- Outline of algorithm:
  - We will be updating a flow f
  - Start with $f = 0$
  - We will maintain a **“residual graph”** G_f
  - A path from s to t in G_f will give us a way to improve our flow.
  - We will continue until there are no s-t paths left in G_f.

Assume for today that we don’t have edges like this, although this assumption can be removed.



Tool: Residual networks

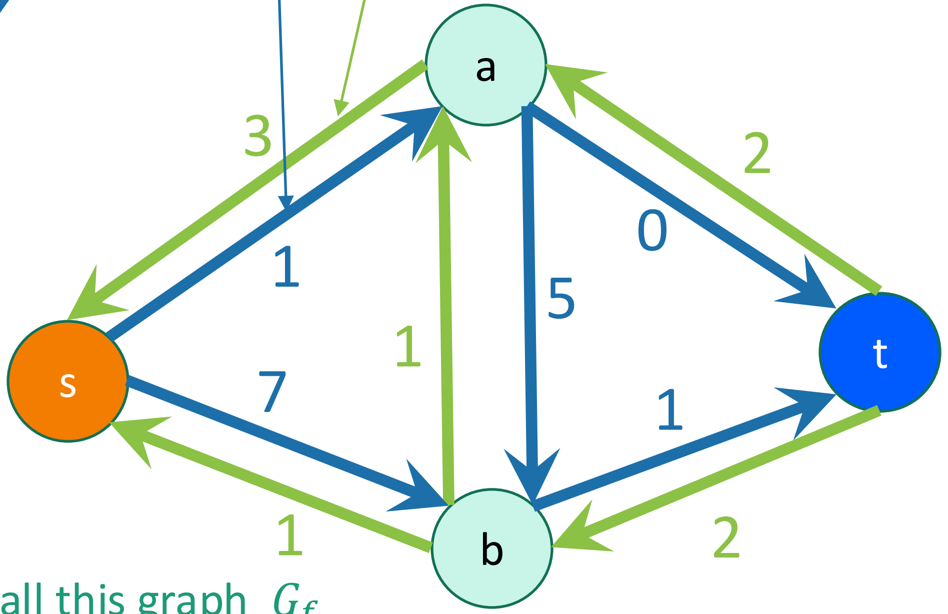
Say we have a flow



Call the flow f
Call the graph G

This forward edge has weight [capacity] – [flow].

This backward edge has weight [flow].

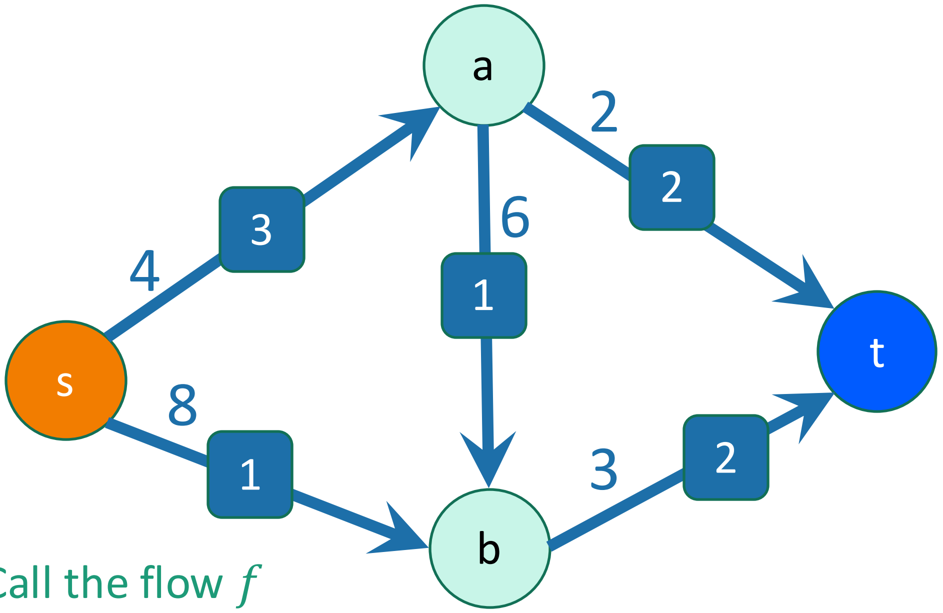


Create a new **residual network** from this flow:

Call this graph G_f

Tool: Residual networks

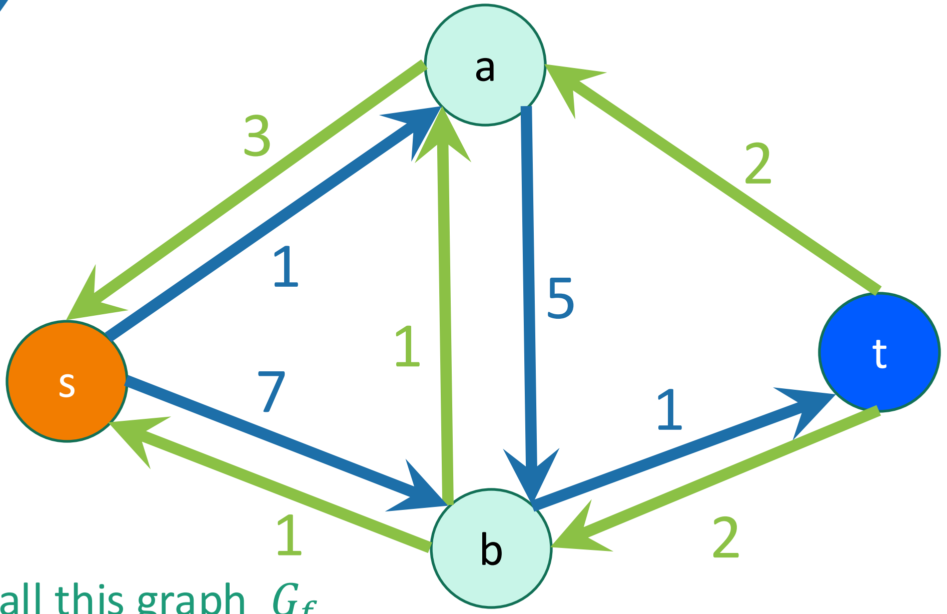
Say we have a flow



Call the flow f
Call the graph G

Forward edges are the amount that's left.
Backwards edges are the amount that's been used.

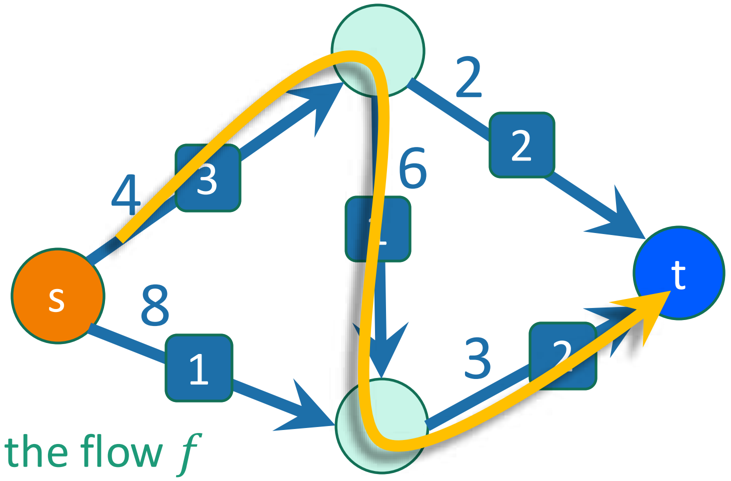
Create a new **residual network** from this flow:



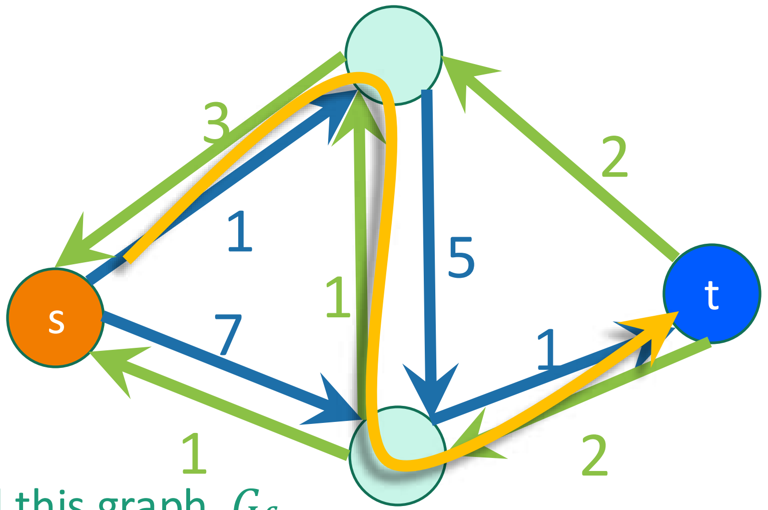
Call this graph G_f

Residual networks tell us how to improve the flow.

- **Definition**: A path from s to t in the residual network is called an **augmenting path**.
- **Claim**: If there is an augmenting path in G_f, we can increase the flow along that path in G.



Call the flow f
Call the graph G

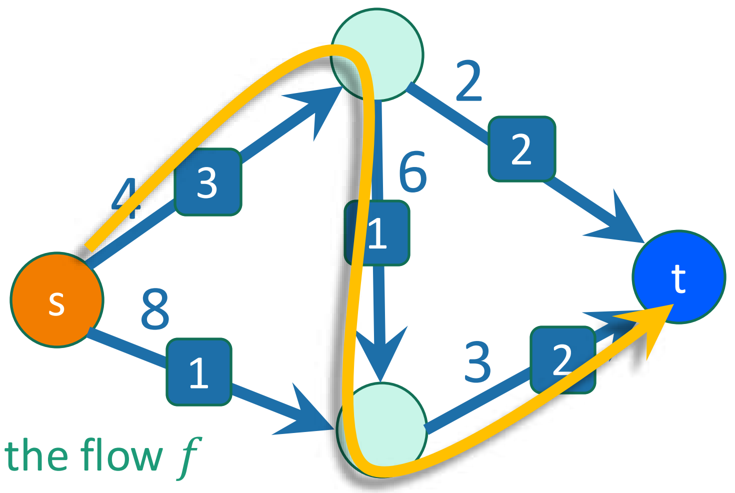


Call this graph G_f

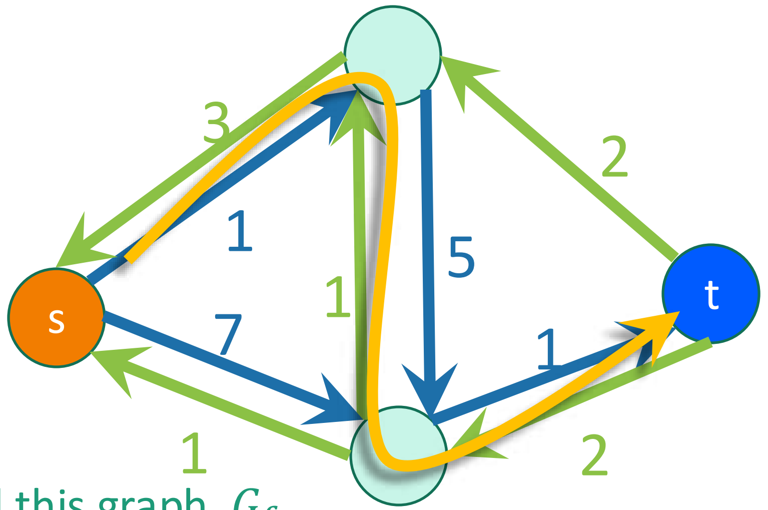
claim:

if there is an augmenting path, we can increase the flow along that path.

- Easy case: every edge on the path in G_f is a **forward edge** in G



Call the flow f
Call the graph G



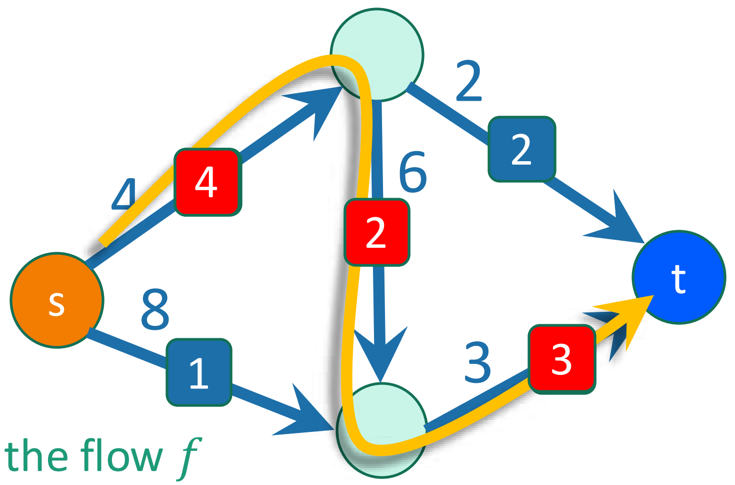
Call this graph G_f

- Forward edges indicate how much stuff can still go through.
- Just increase the flow on all the edges!

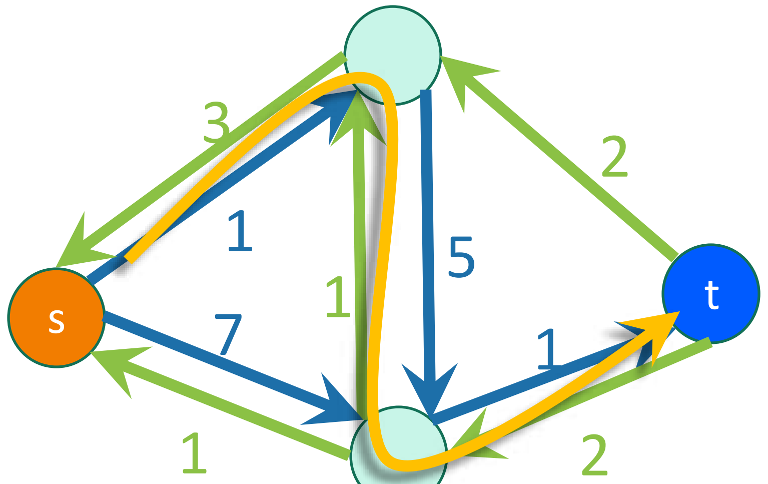
claim:

if there is an augmenting path, we can increase the flow along that path.

- Easy case: every edge on the path in G_f is a **forward edge** in G



Call the flow f
Call the graph G



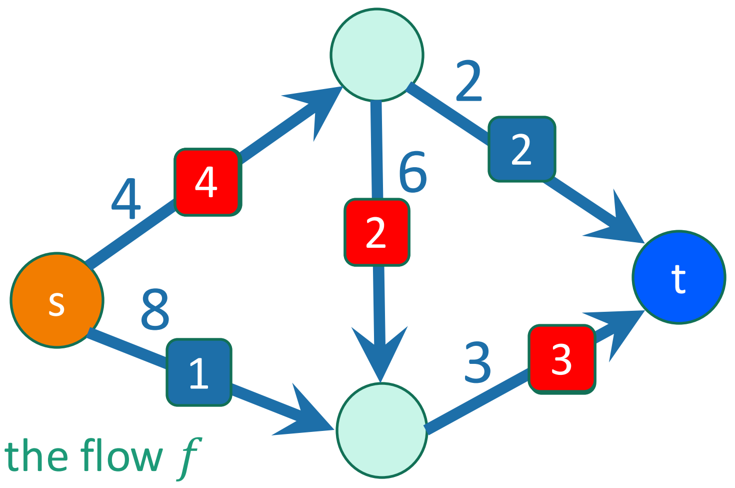
Call this graph G_f

- Forward edges indicate how much stuff can still go through.
- Just increase the flow on all the edges!

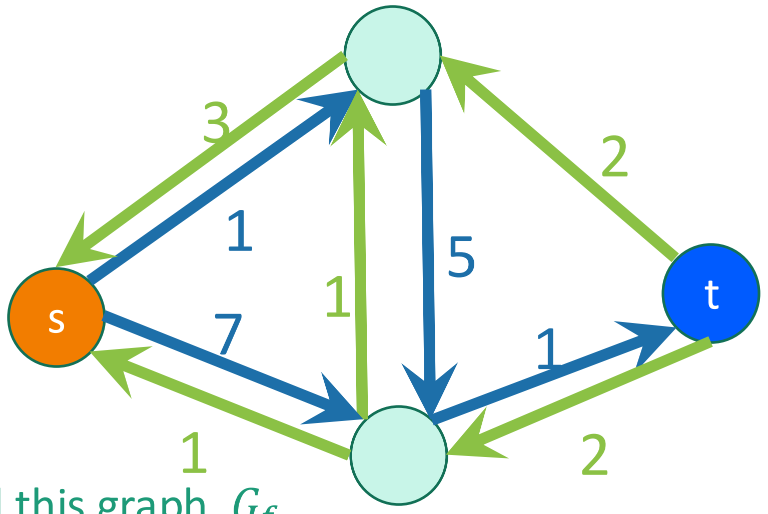
claim:

if there is an augmenting path, we can increase the flow along that path.

- Easy case: every edge on the path in G_f is a **forward edge** in G



Call the flow f
Call the graph G



Call this graph G_f

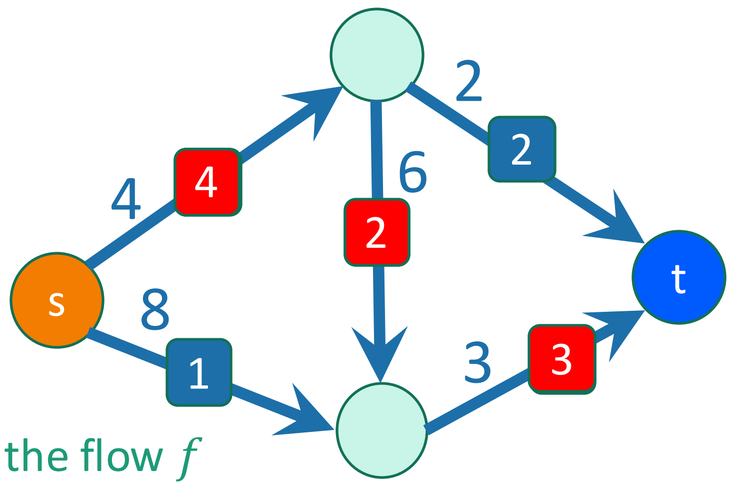
- Forward edges indicate how much stuff can still go through.
- Just increase the flow on all the edges!

Then update the residual graph.

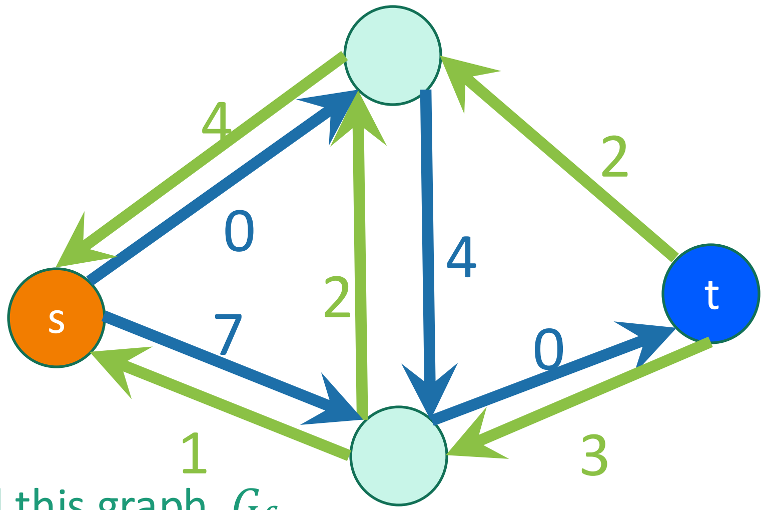
claim:

if there is an augmenting path, we can increase the flow along that path.

- Easy case: every edge on the path in G_f is a **forward edge** in G.



Call the flow f
Call the graph G



Call this graph G_f

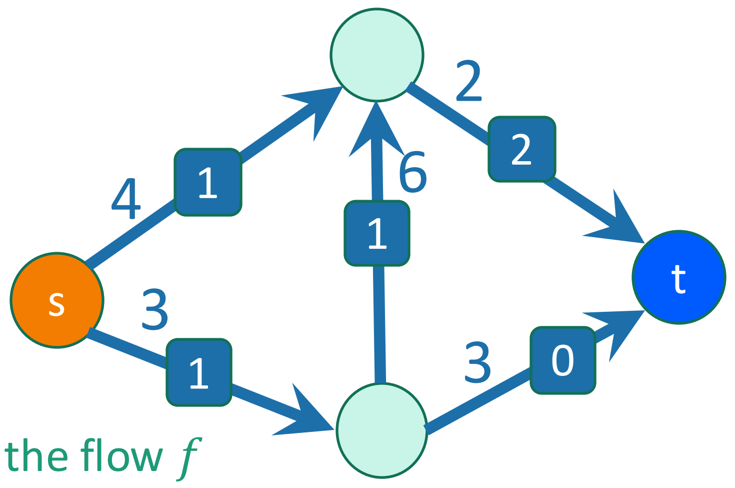
- Forward edges indicate how much stuff can still go through.
- Just increase the flow on all the edges!

Then update the residual graph.

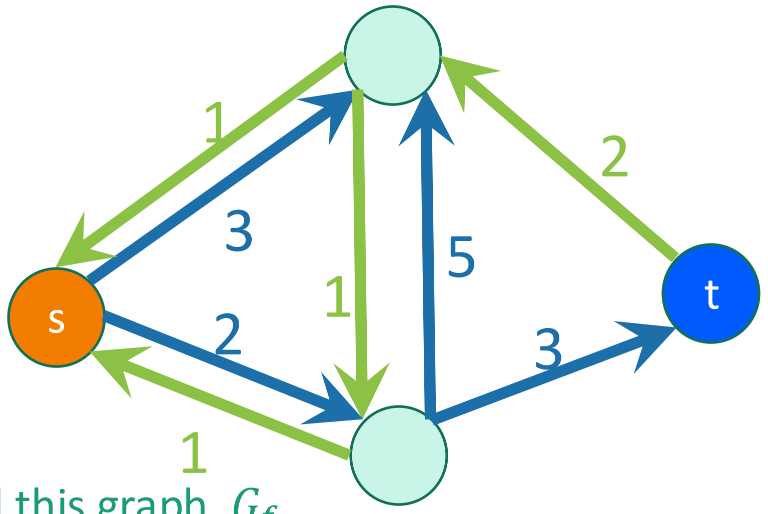
claim:

if there is an augmenting path, we can increase the flow along that path.

- Harder case: there are **backward edges** in G in the path.
  - Here's a slightly different example of a flow:



Call the flow f
Call the graph G



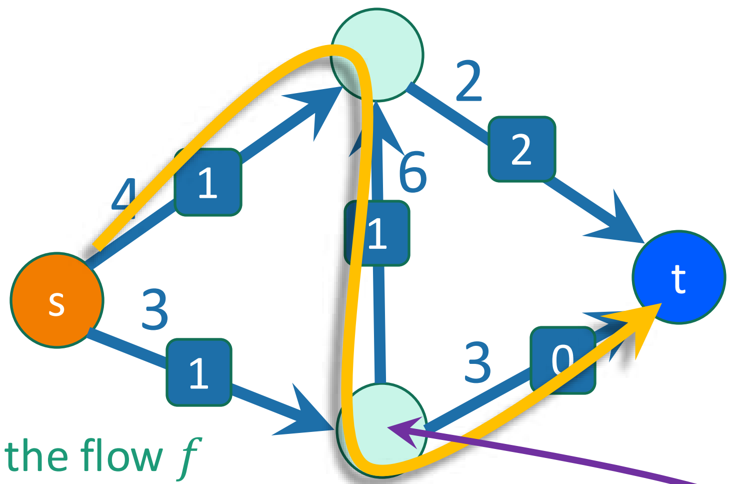
Call this graph G_f

I changed some of the weights and edge directions.

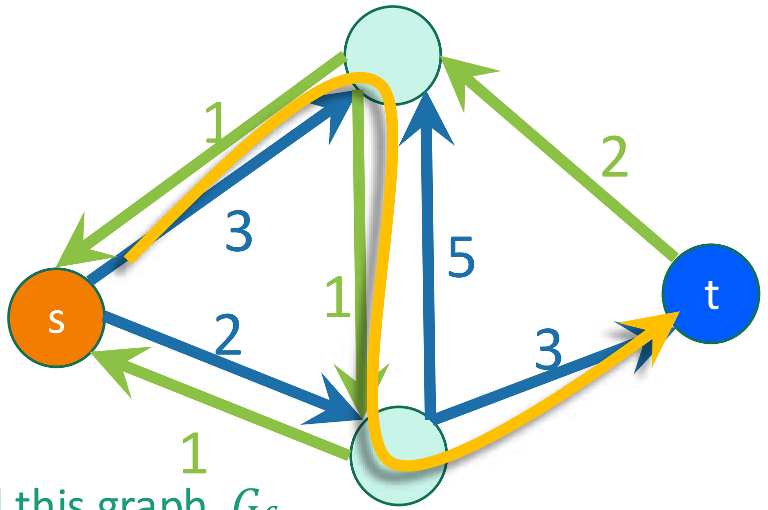
claim:

if there is an augmenting path, we can increase the flow along that path.

- Harder case: there are **backward edges** in G in the path.
  - Here's a slightly different example of a flow:



Call the flow f
Call the graph G



Call this graph G_f

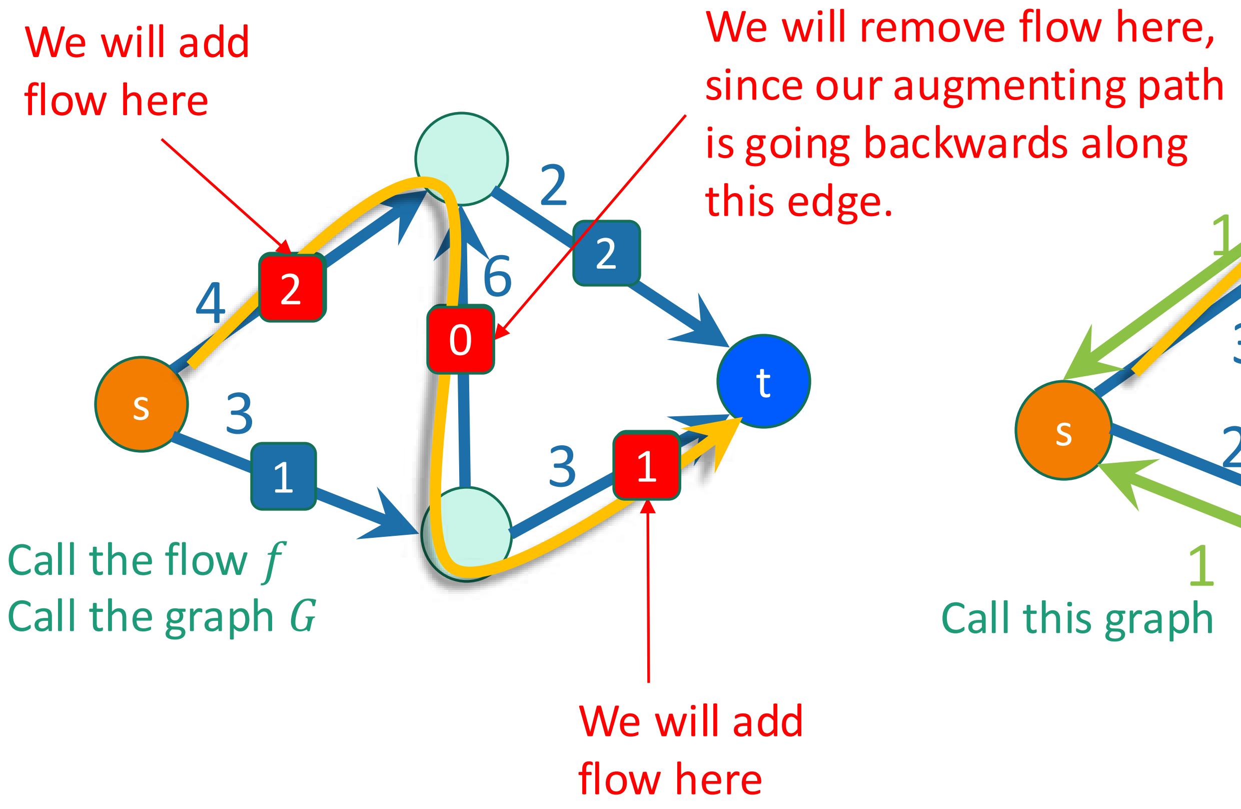
Now we should NOT increase the flow at all the edges along the path!

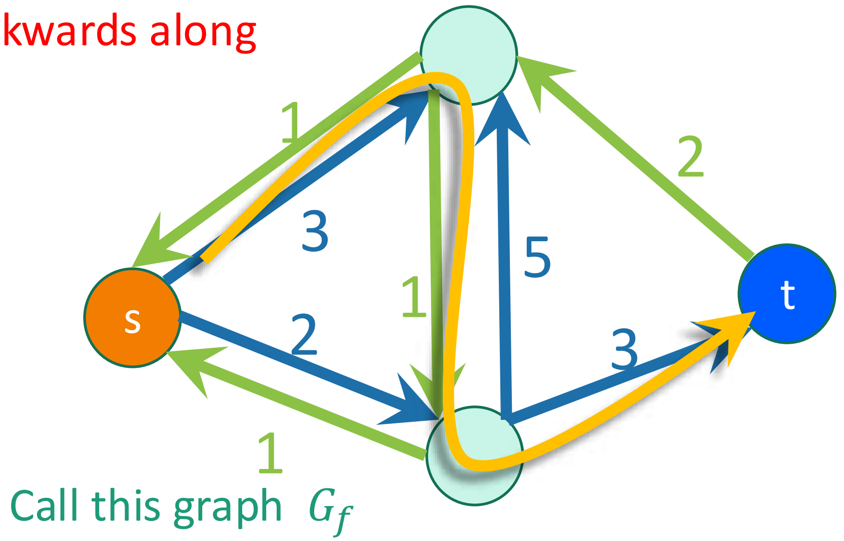
- For example, that will mess up the conservation of stuff at this vertex.

I changed some of the weights and edge directions.

claim:
if there is an augmenting path, we can increase the flow along that path.

- In this case we do something a bit different:



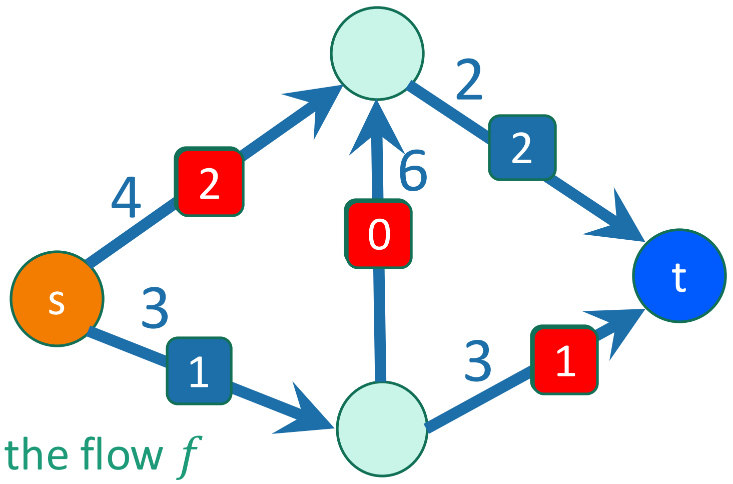


claim:

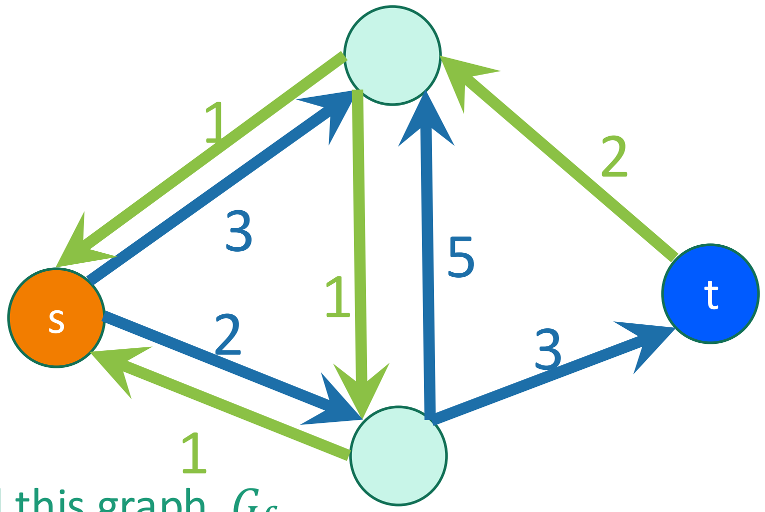
if there is an augmenting path, we can increase the flow along that path.

- In this case we do something a bit different:

Then we'll update the residual graph:



Call the flow f
Call the graph G



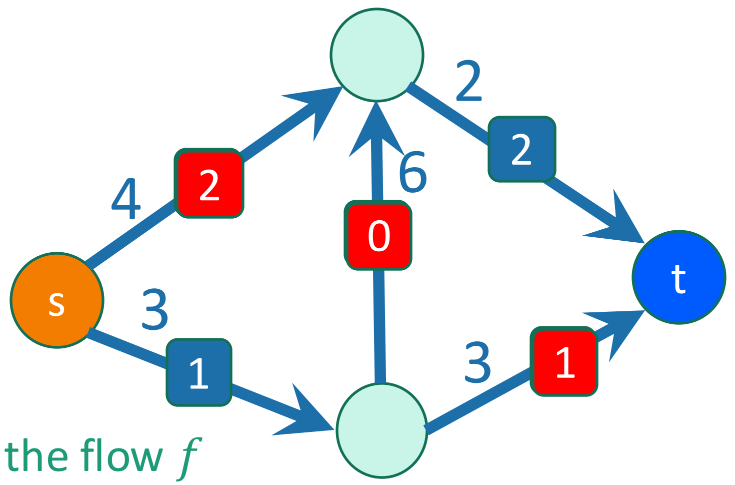
Call this graph G_f

claim:

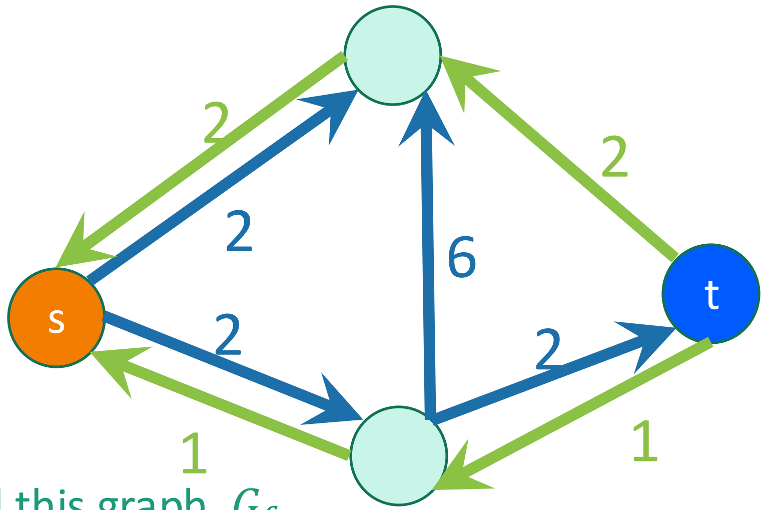
if there is an augmenting path, we can increase the flow along that path.

- In this case we do something a bit different:

Then we'll update the residual graph:

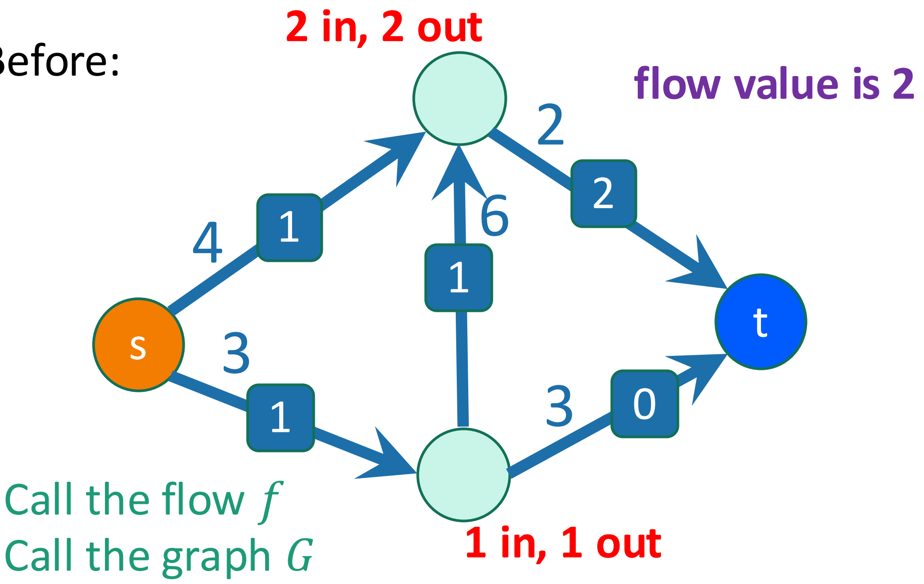


Call the flow f
Call the graph G

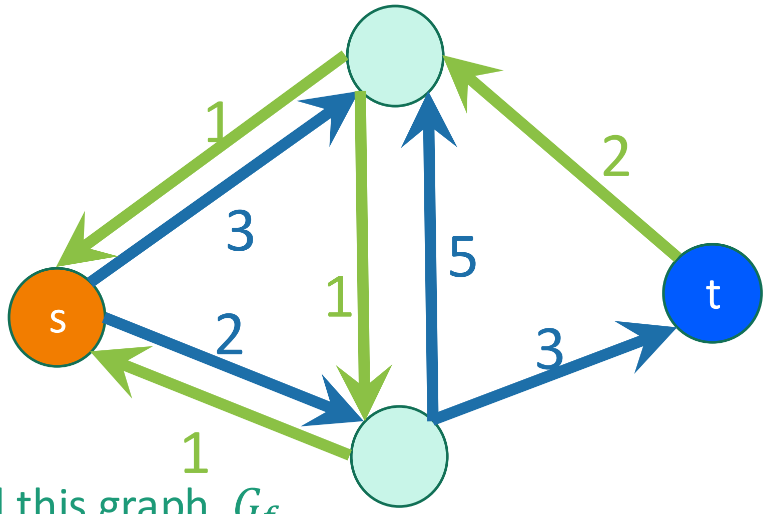


Call this graph G_f

Before:

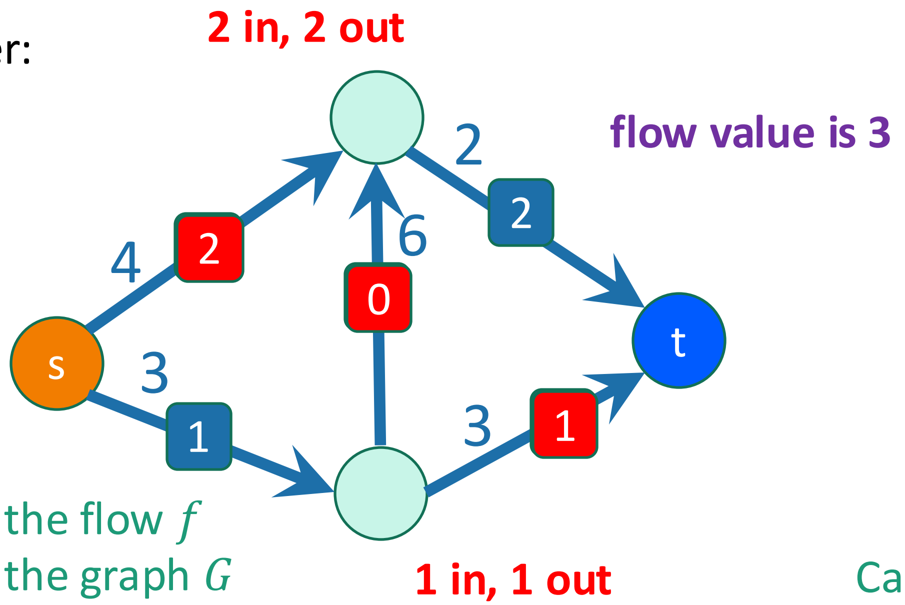


Call the flow f
Call the graph G

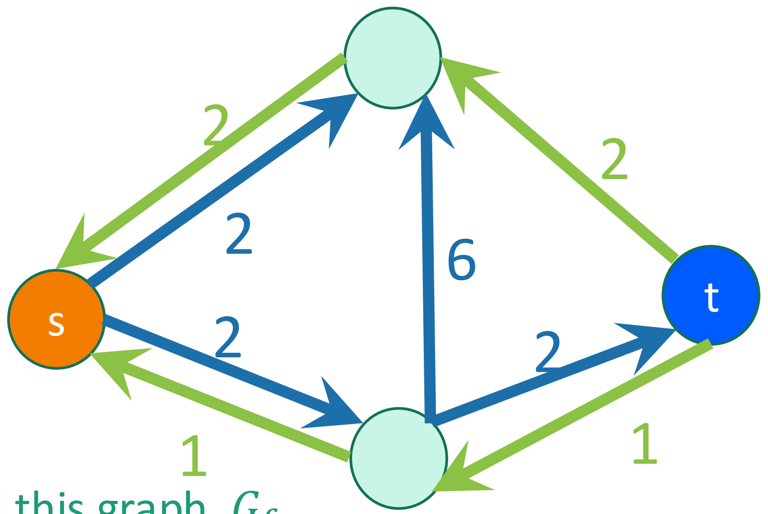


Call this graph G_f

After:



Call the flow f
Call the graph G



Call this graph G_f

Still a legit flow, but with a bigger value!

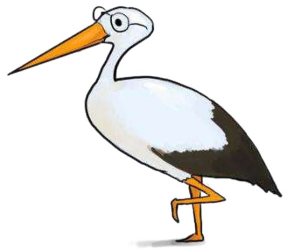
claim:

if there is an augmenting path, we can increase the flow along that path.

proof:

```
• increaseFlow(path P in $G_f$ , flow $f$ ):
    • x = min weight on any edge in P
    • for (u,v) in P:
        • if (u,v) in E, $f'(u,v) \leftarrow f(u,v) + x$.
        • if (v,u) in E, $f'(v,u) \leftarrow f(v,u) - x$
    • return $f'$
```

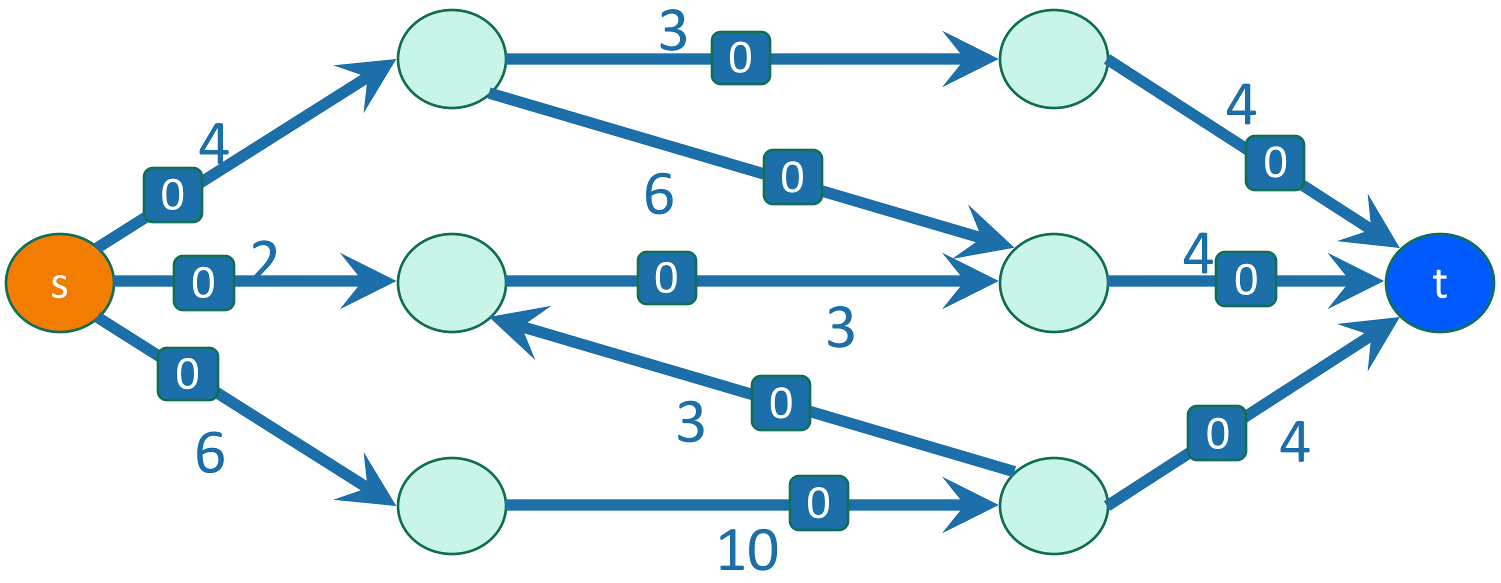
Check that this always makes a bigger (and legit) flow!

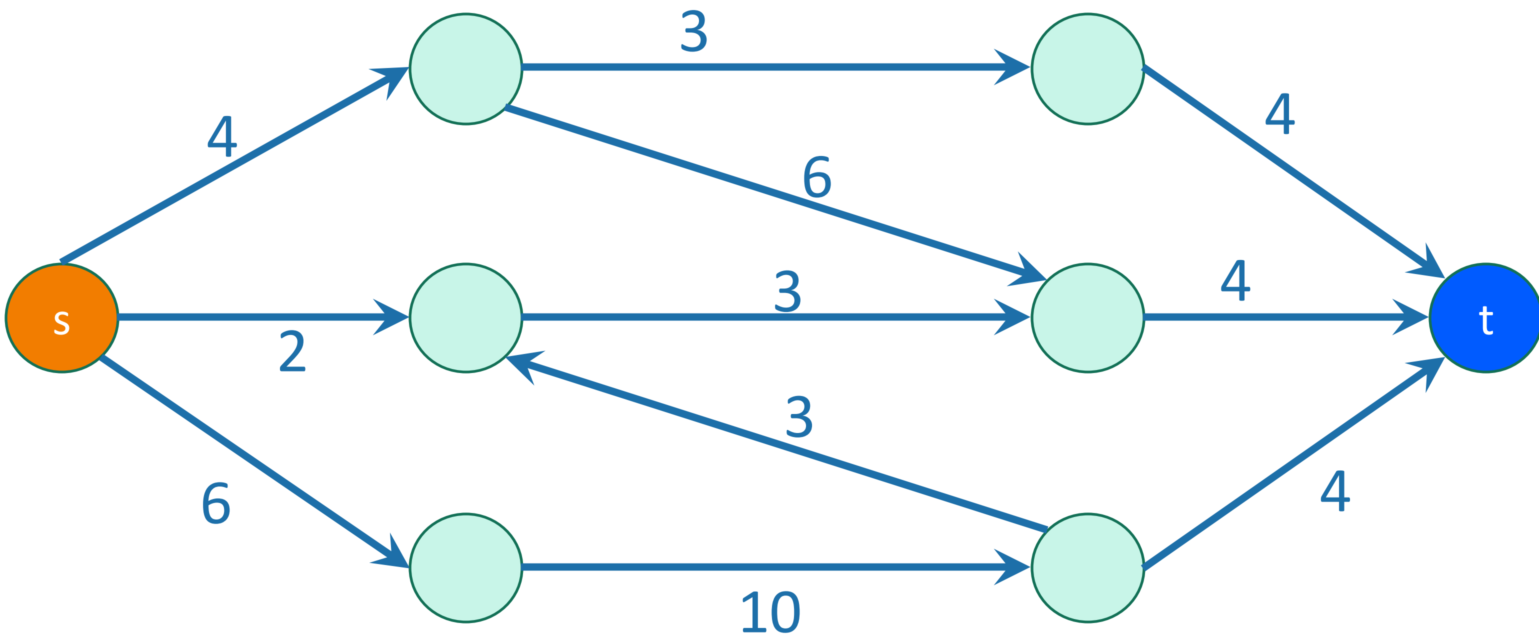


Ford-Fulkerson Algorithm

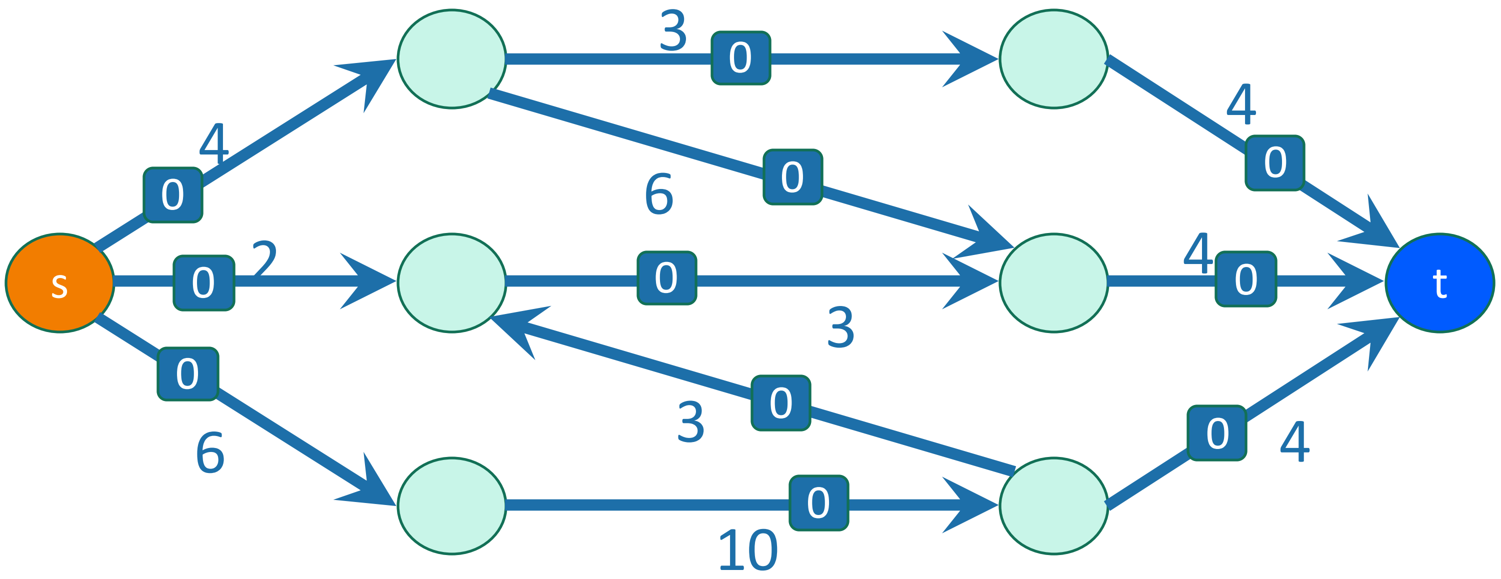
```
• **Ford-Fulkerson**(G):
    • $f \leftarrow$ all zero flow.
    • $G_f \leftarrow G$
    • **while** t is reachable from s in $G_f$
        • Find a path P from s to t in $G_f$        // eg, use BFS
        • $f \leftarrow$ **increaseFlow**($P$,$f$)
        • update $G_f$
    • **return** $f$
```

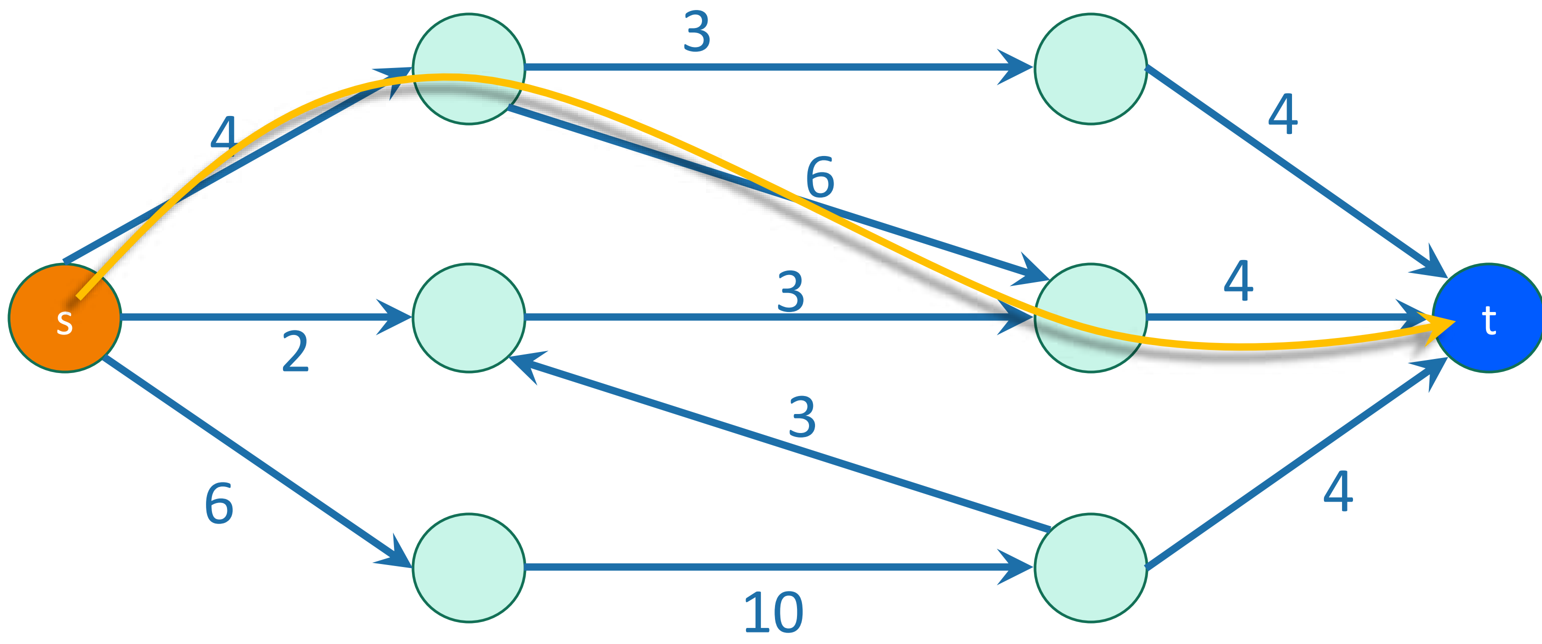
Example of Ford-Fulkerson



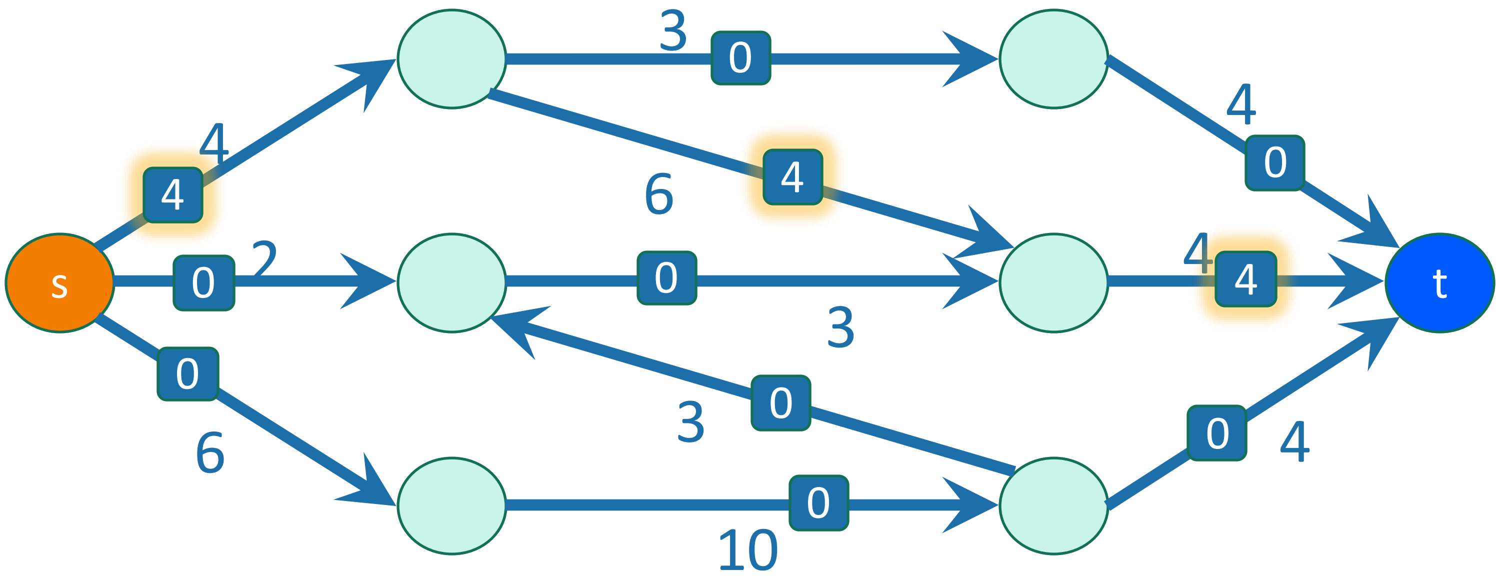


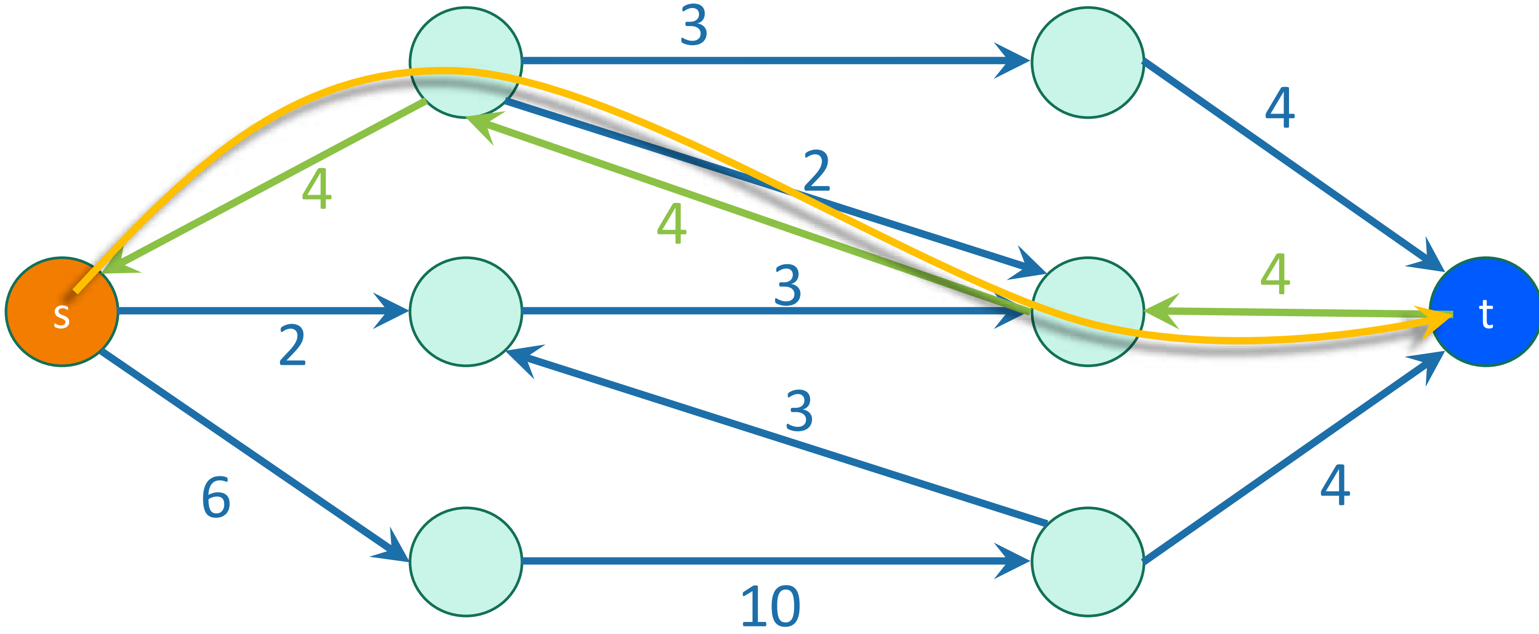
Example of Ford-Fulkerson



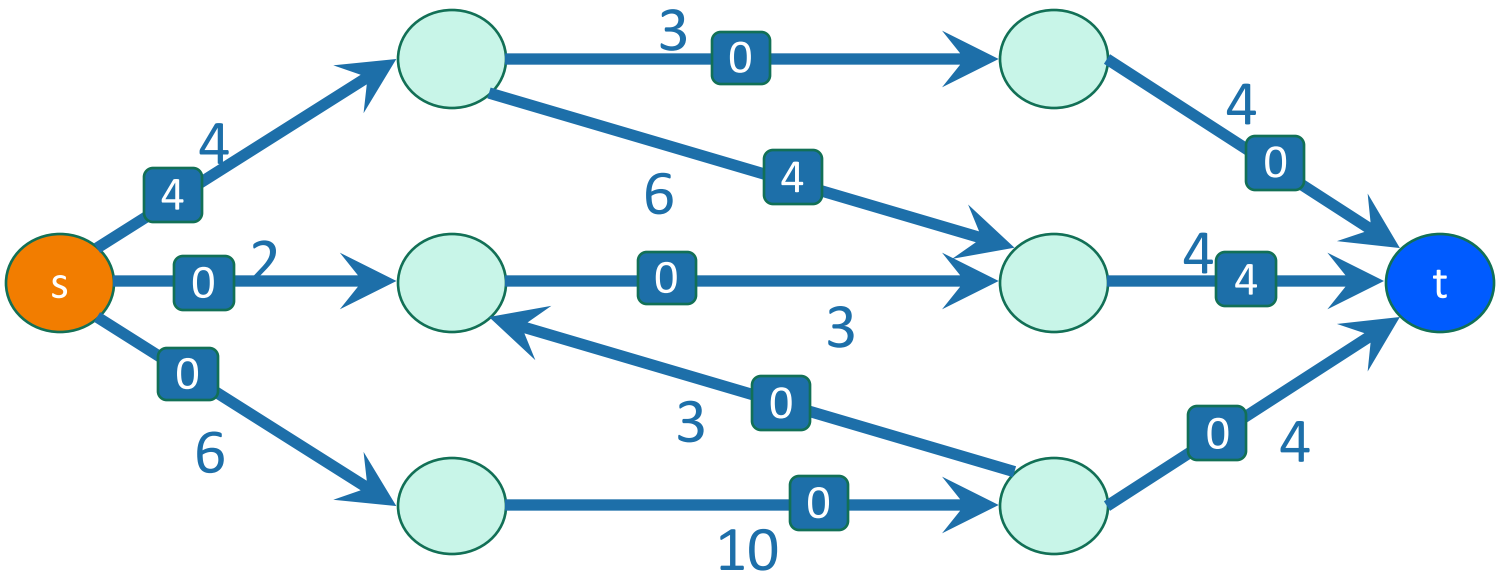


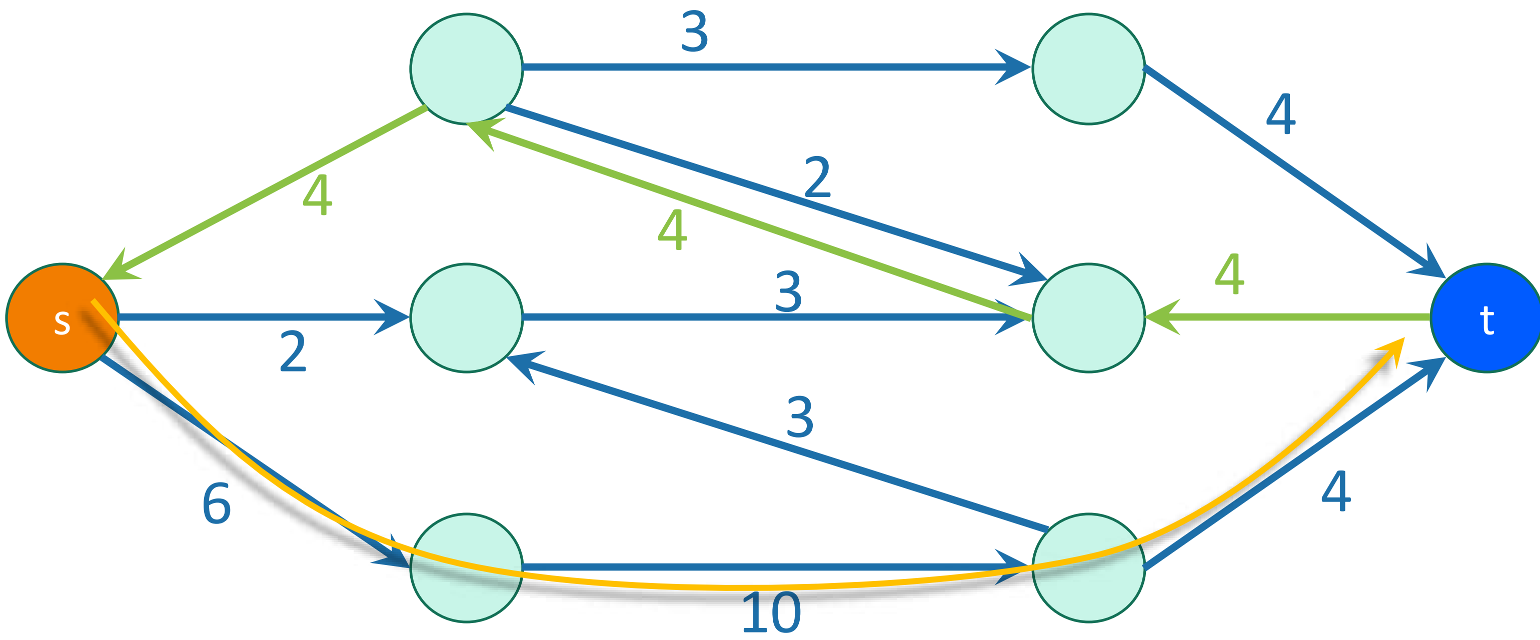
Example of Ford-Fulkerson



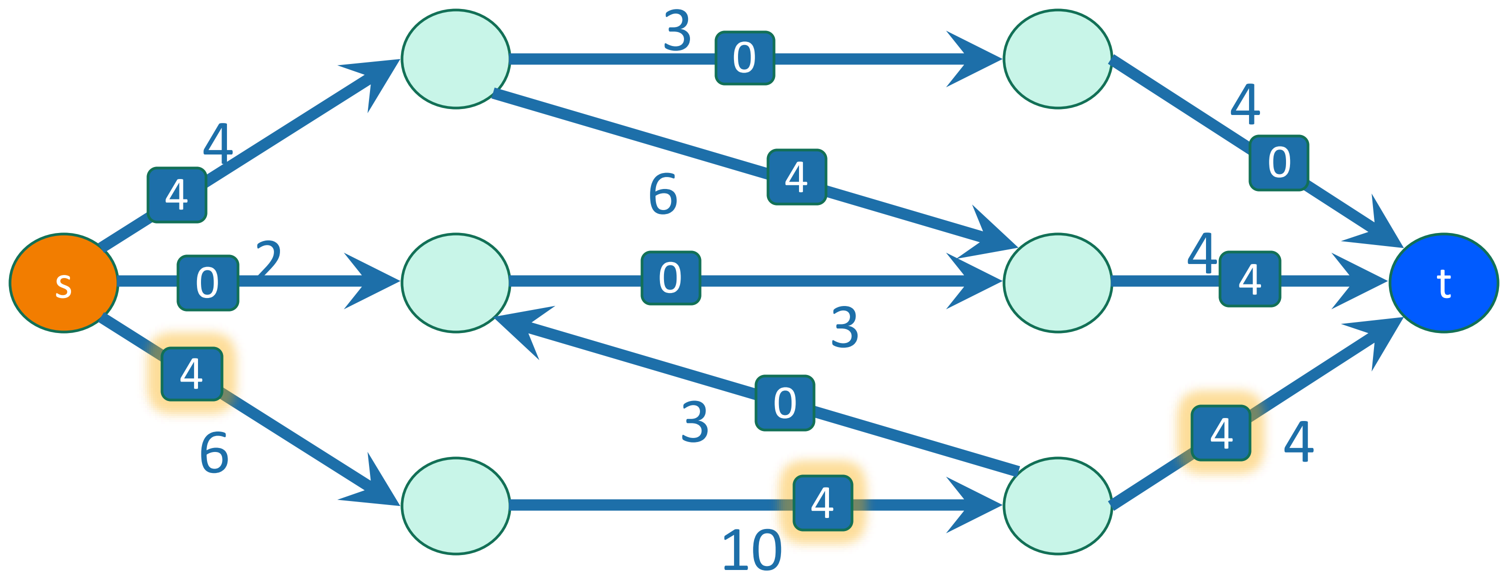


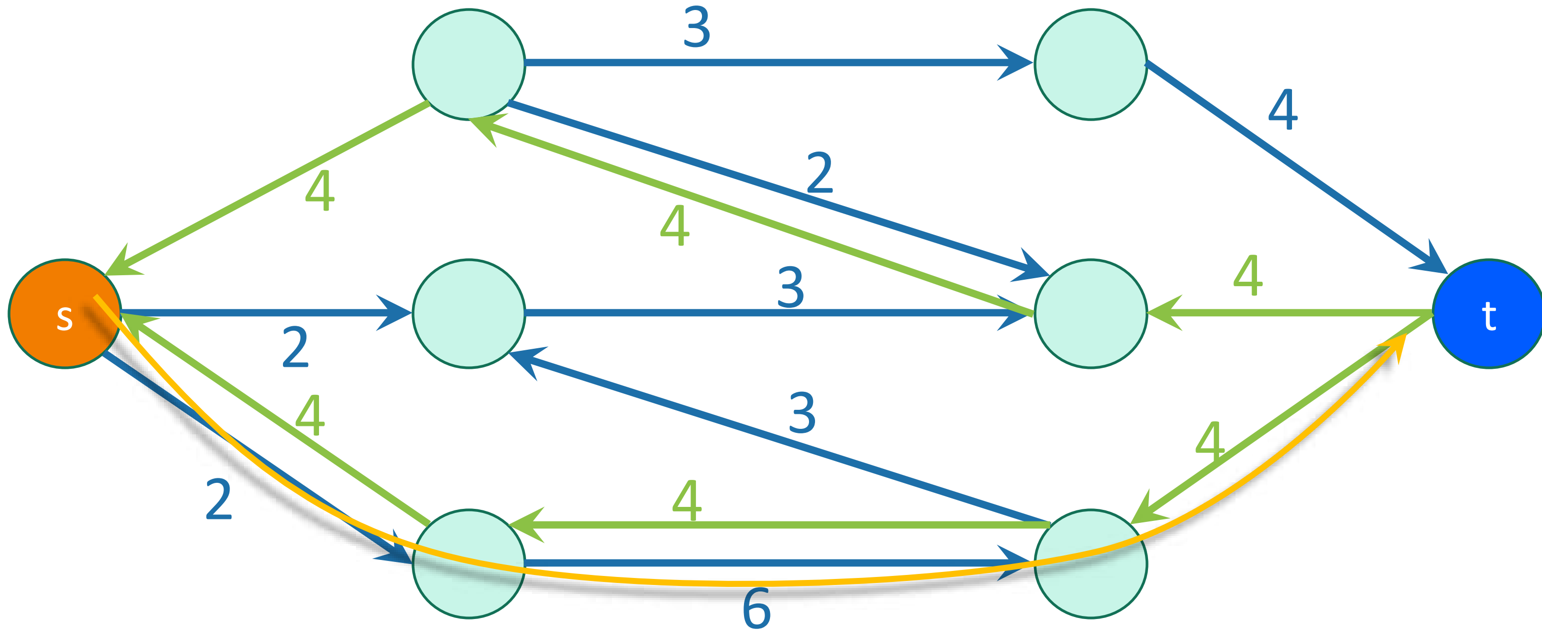
Example of Ford-Fulkerson





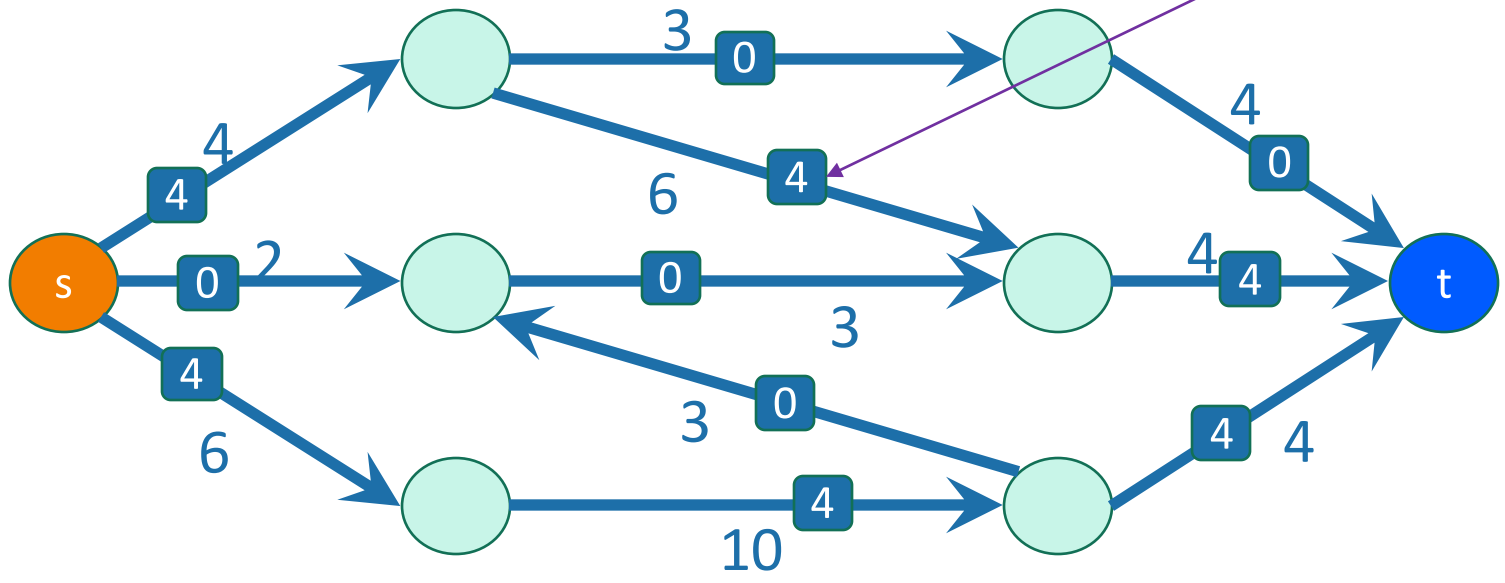
Example of Ford-Fulkerson



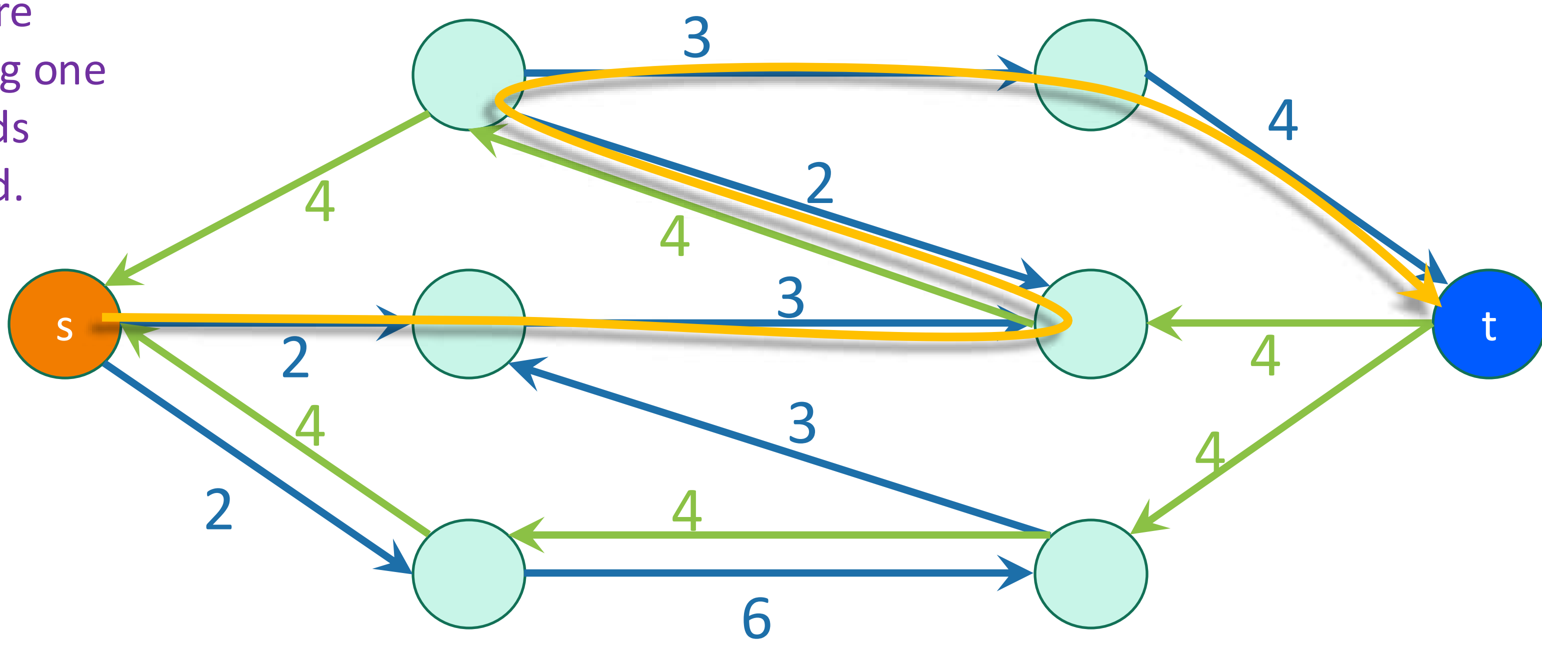


Example of Ford-Fulkerson

We will **remove** flow from this edge.

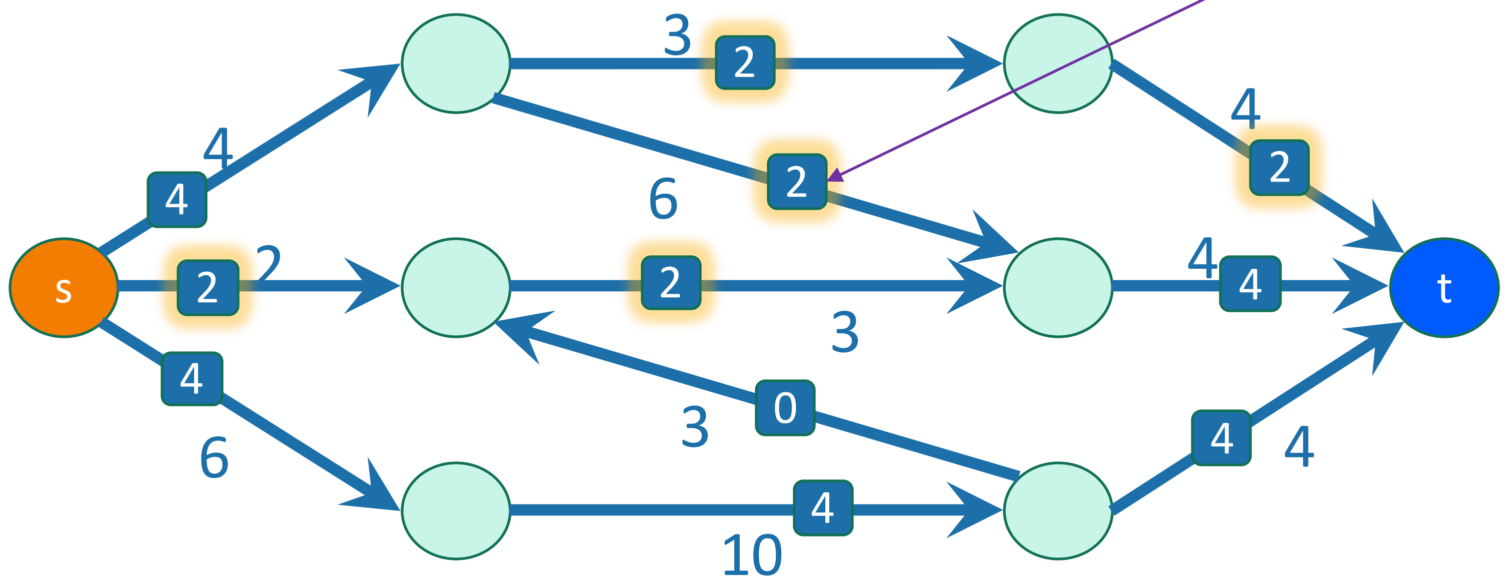


Notice that we're going back along one of the backwards edges we added.

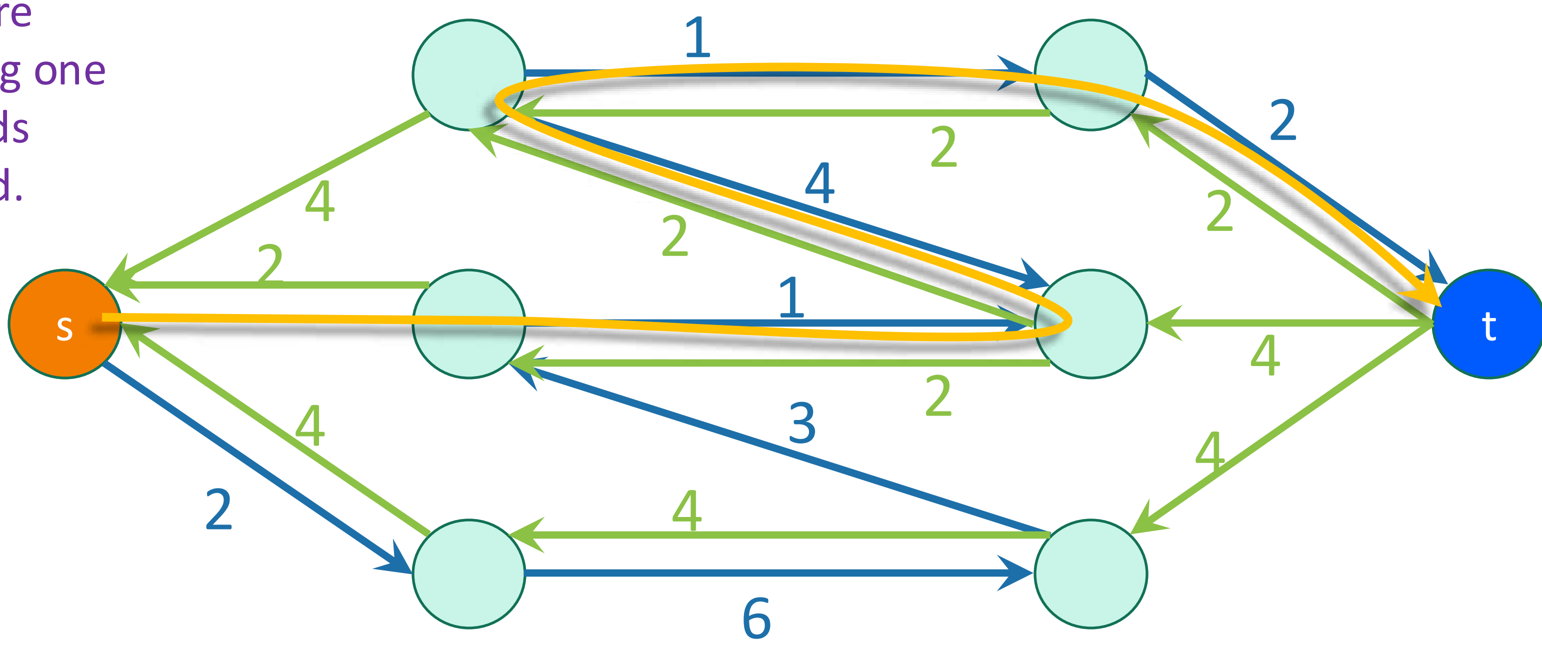


Example of Ford-Fulkerson

We will **remove** flow from this edge.

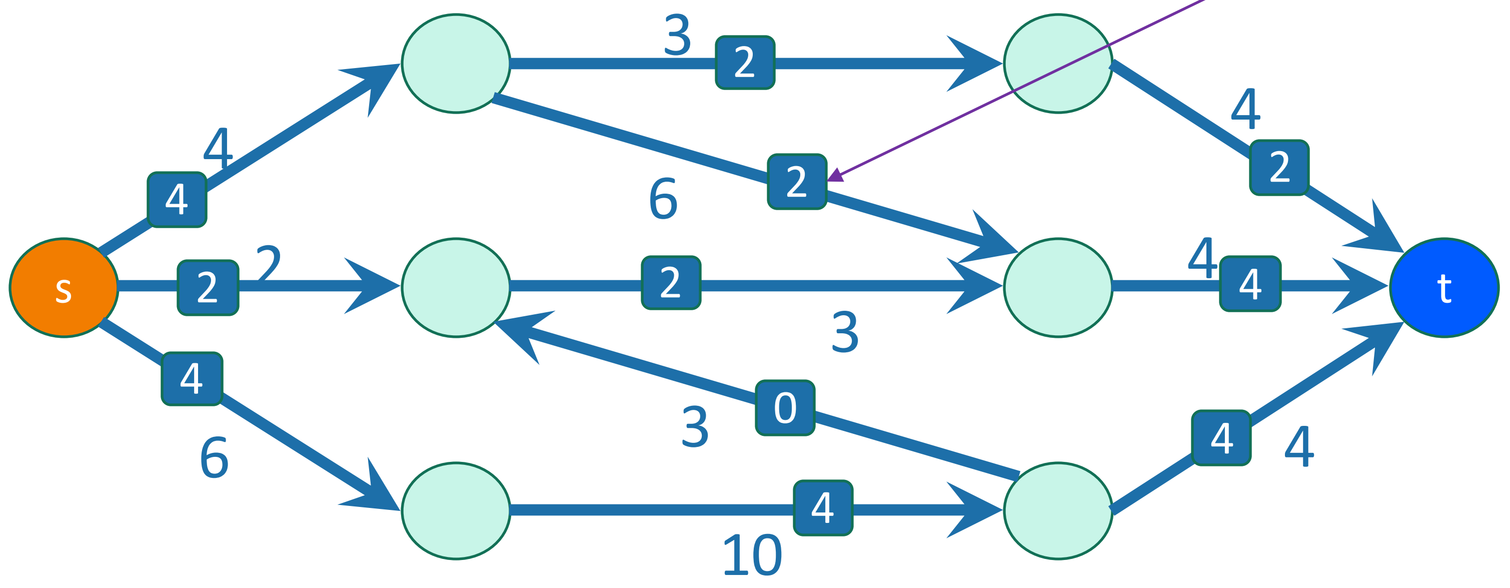


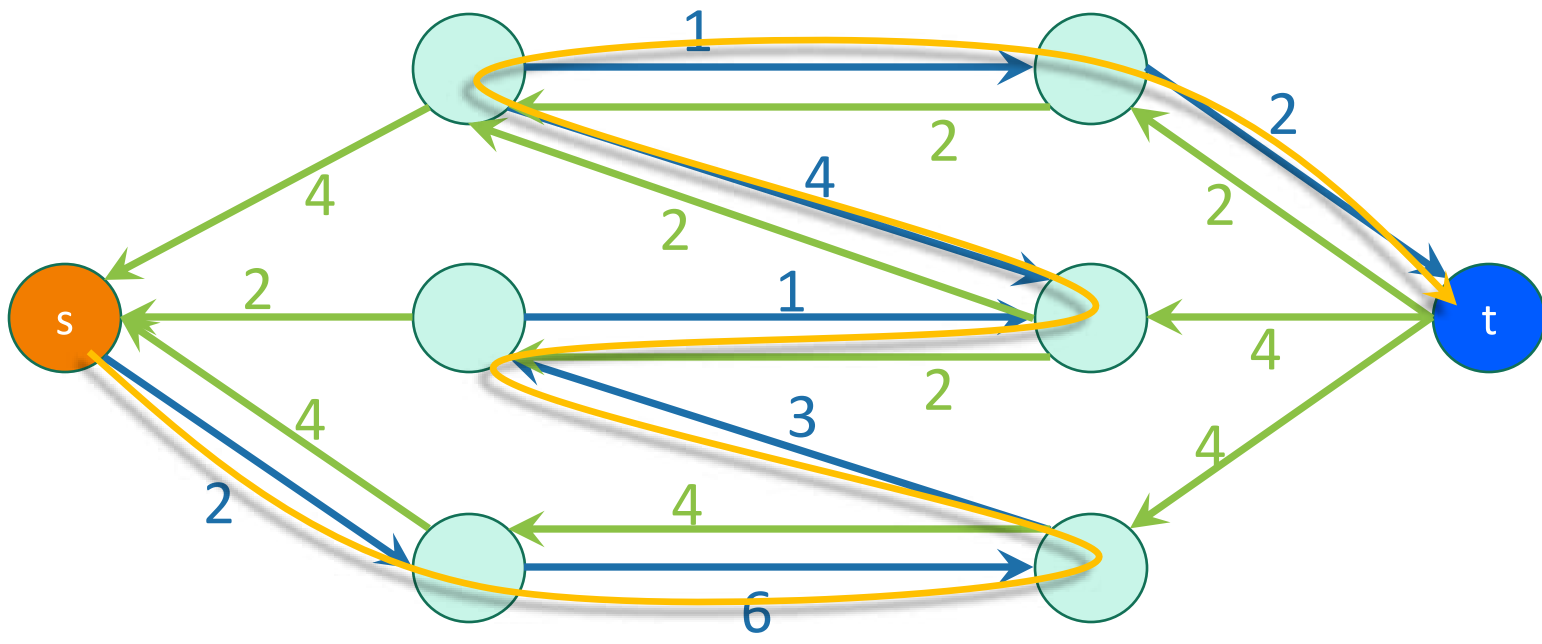
Notice that we're going back along one of the backwards edges we added.



Example of Ford-Fulkerson

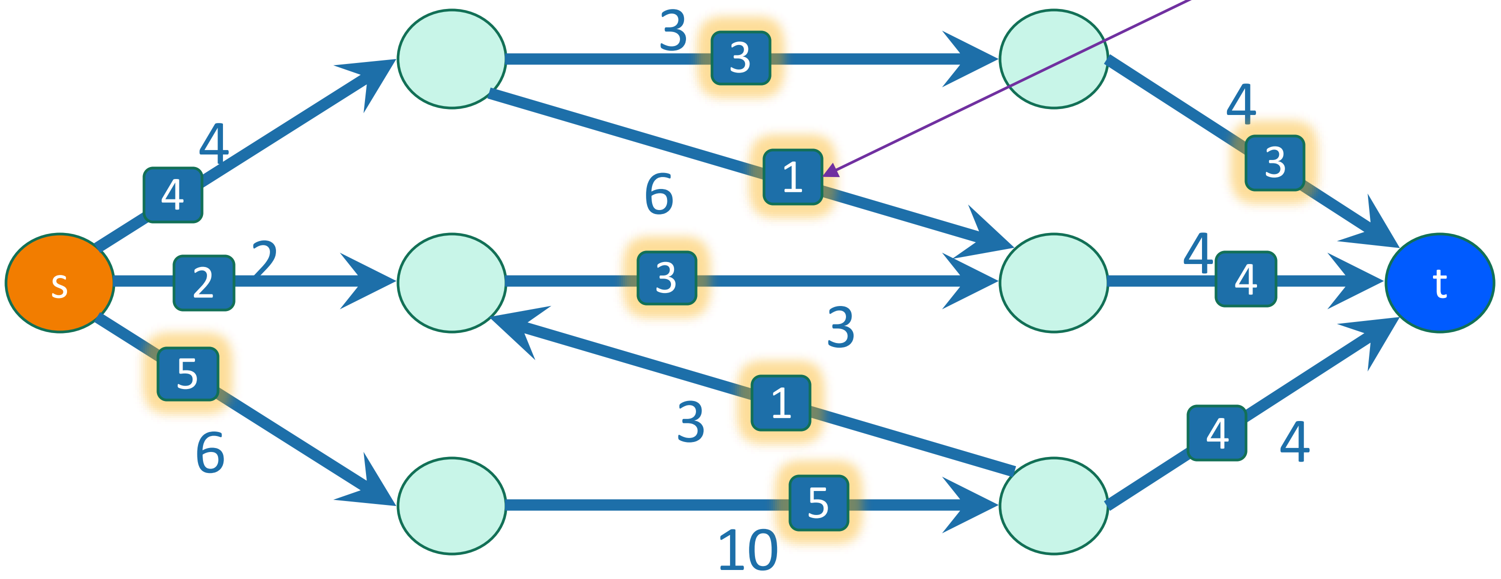
We will remove flow from this edge AGAIN.

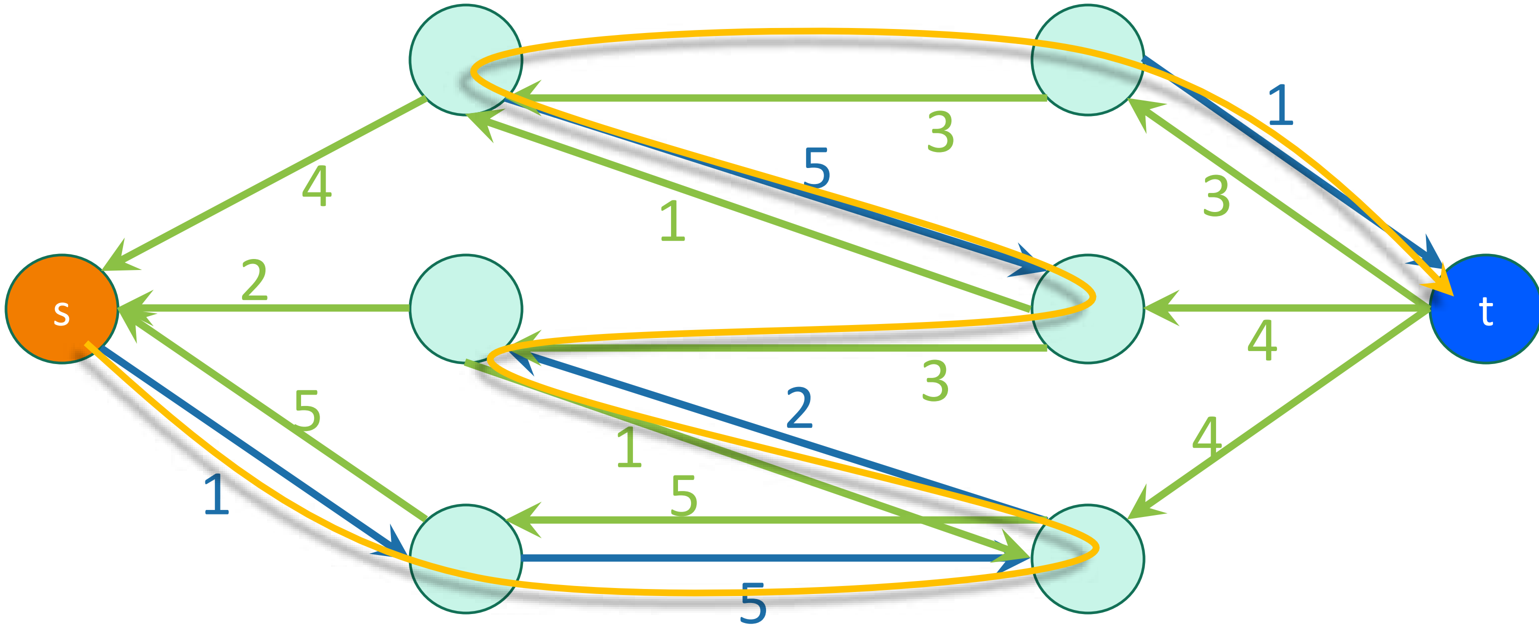




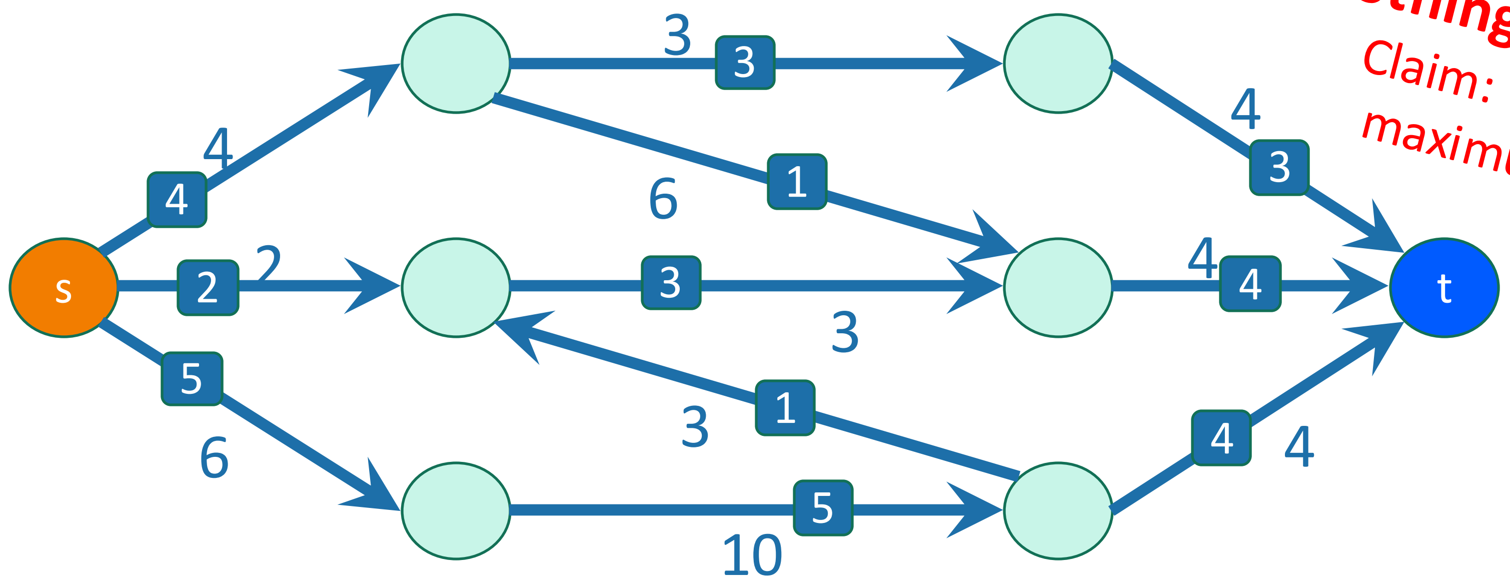
Example of Ford-Fulkerson

We will remove flow from this edge AGAIN.



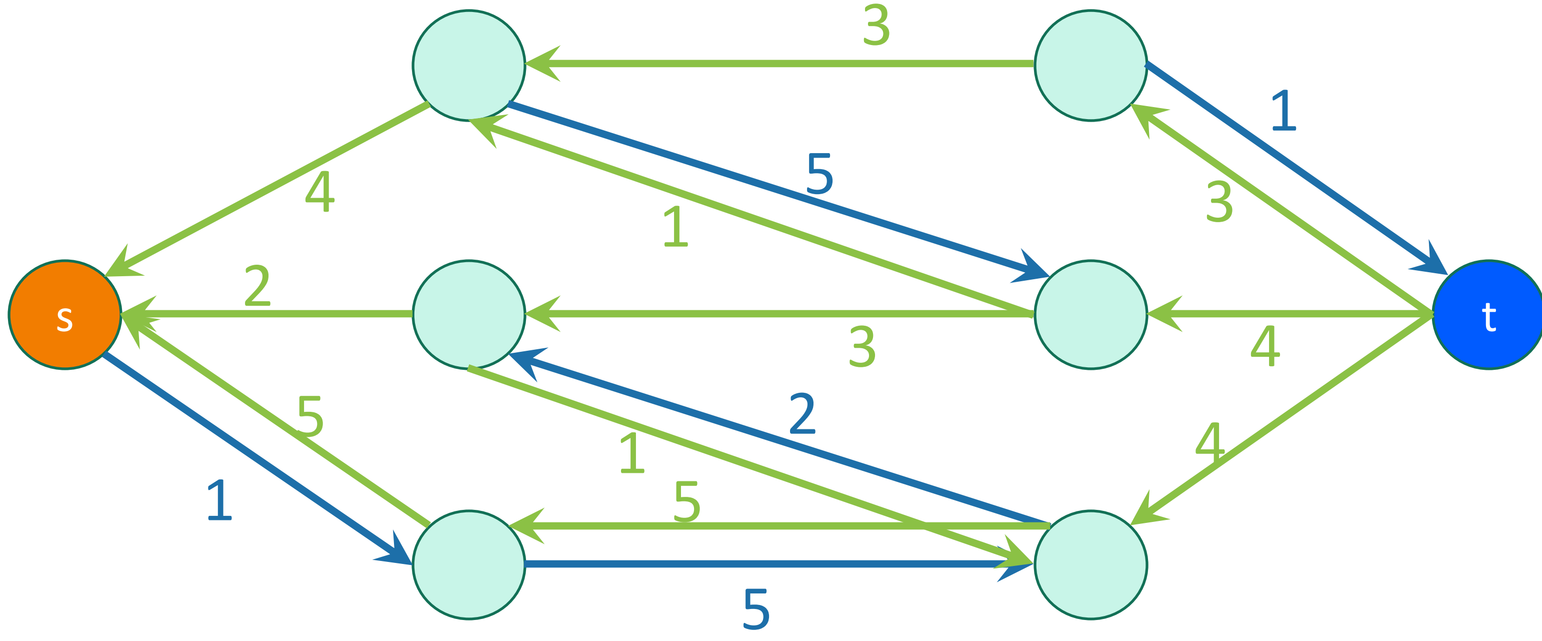


Example of Ford-Fulkerson



Now we have nothing left to do!

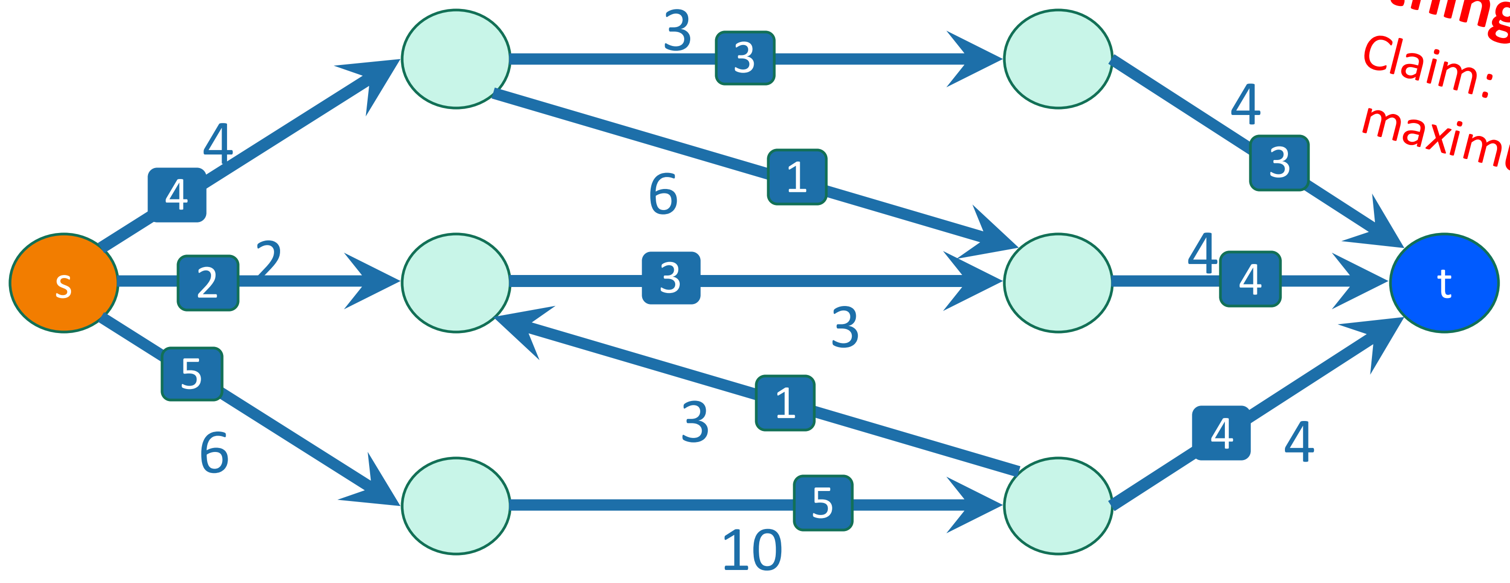
Claim: This is the maximum flow.

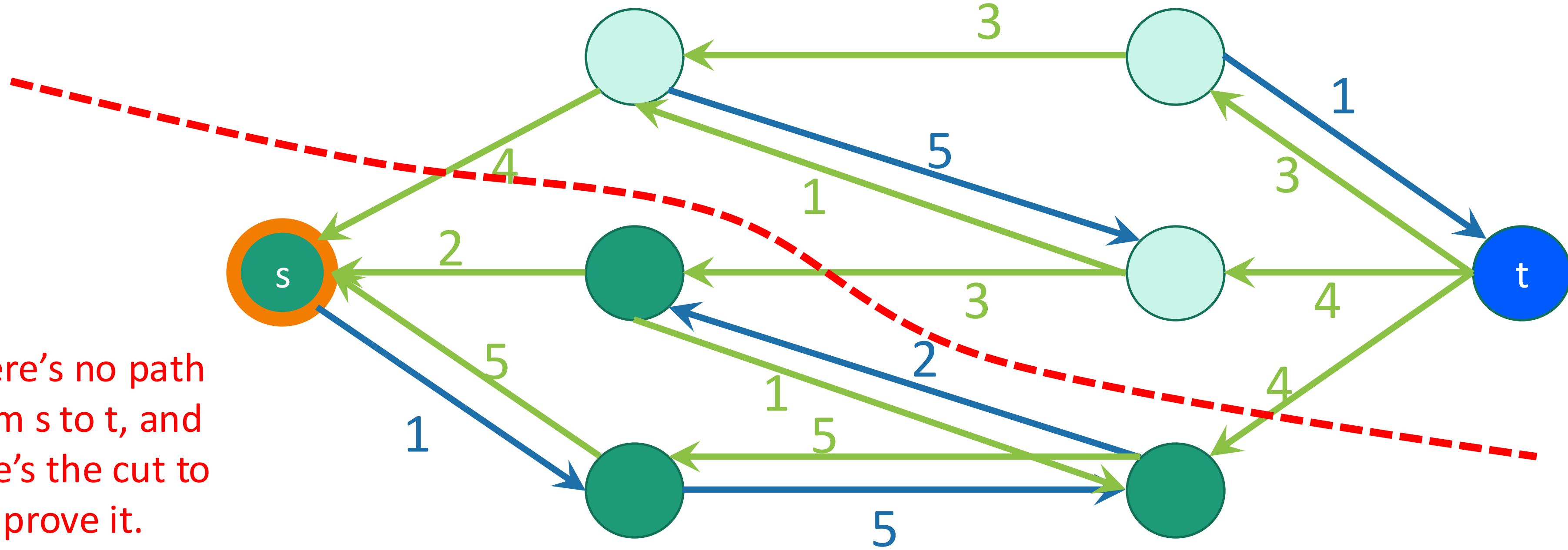


Example of Ford-Fulkerson

Now we have nothing left to do!

Claim: This is the maximum flow.



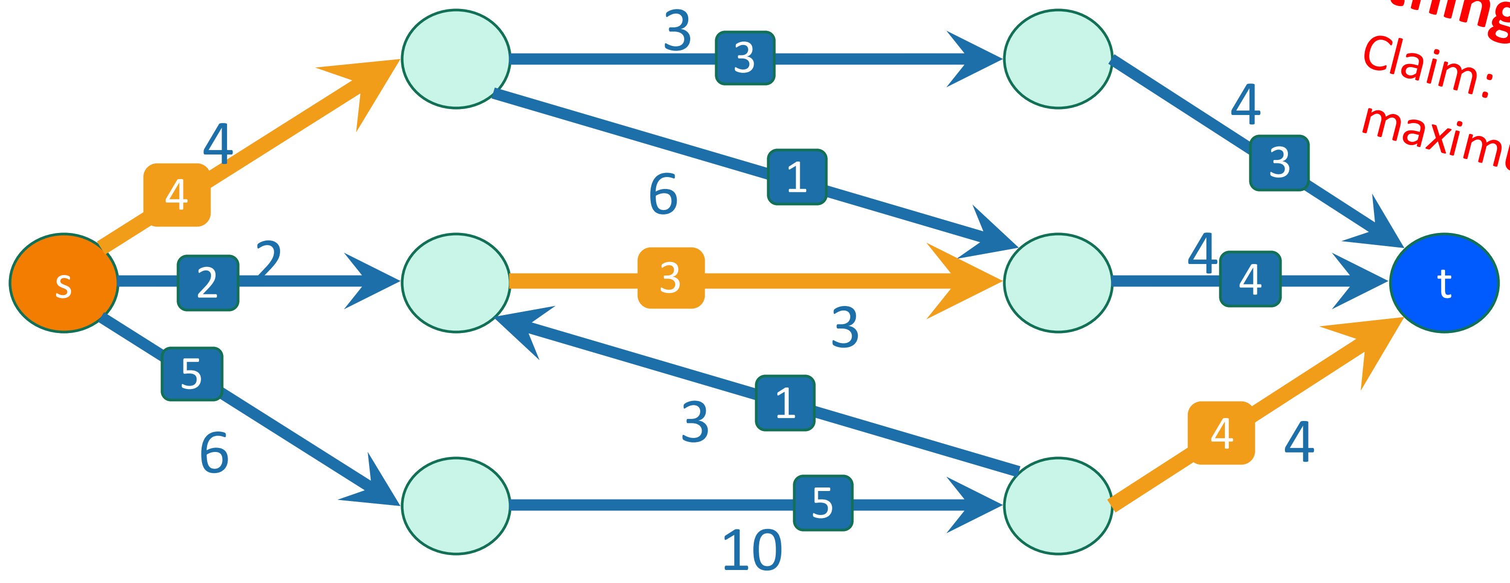


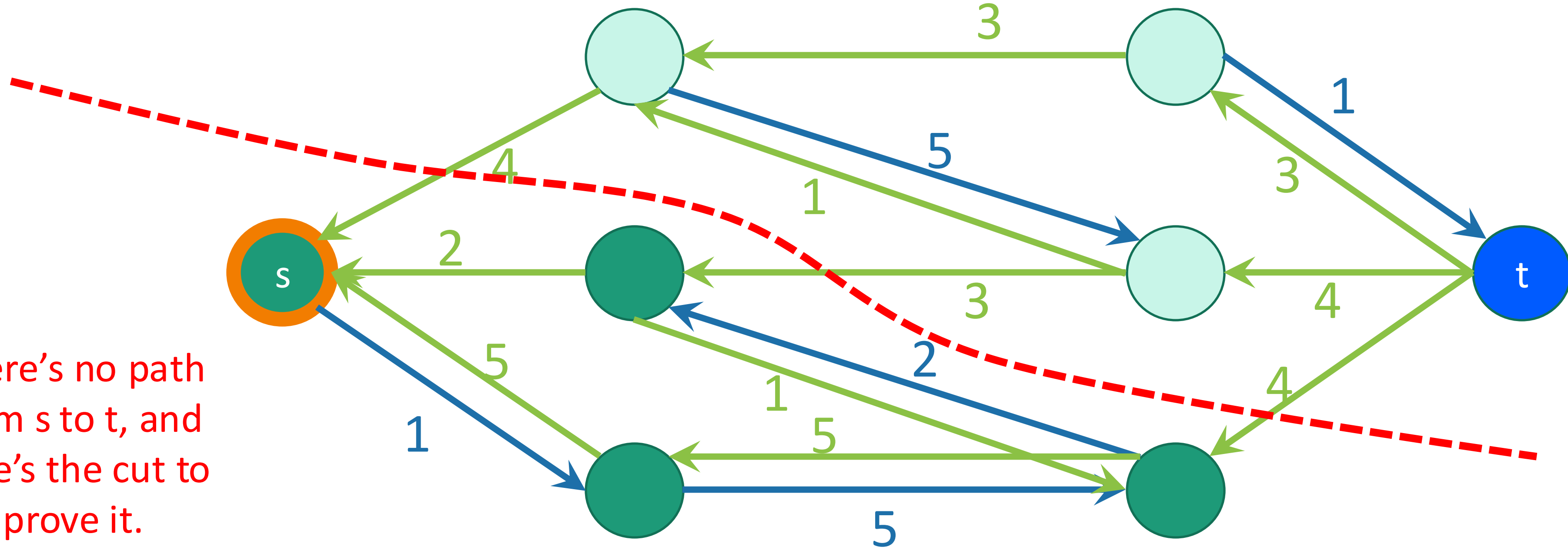
There's no path from s to t, and here's the cut to prove it.

Example of Ford-Fulkerson

Now we have nothing left to do!

Claim: This is the maximum flow.



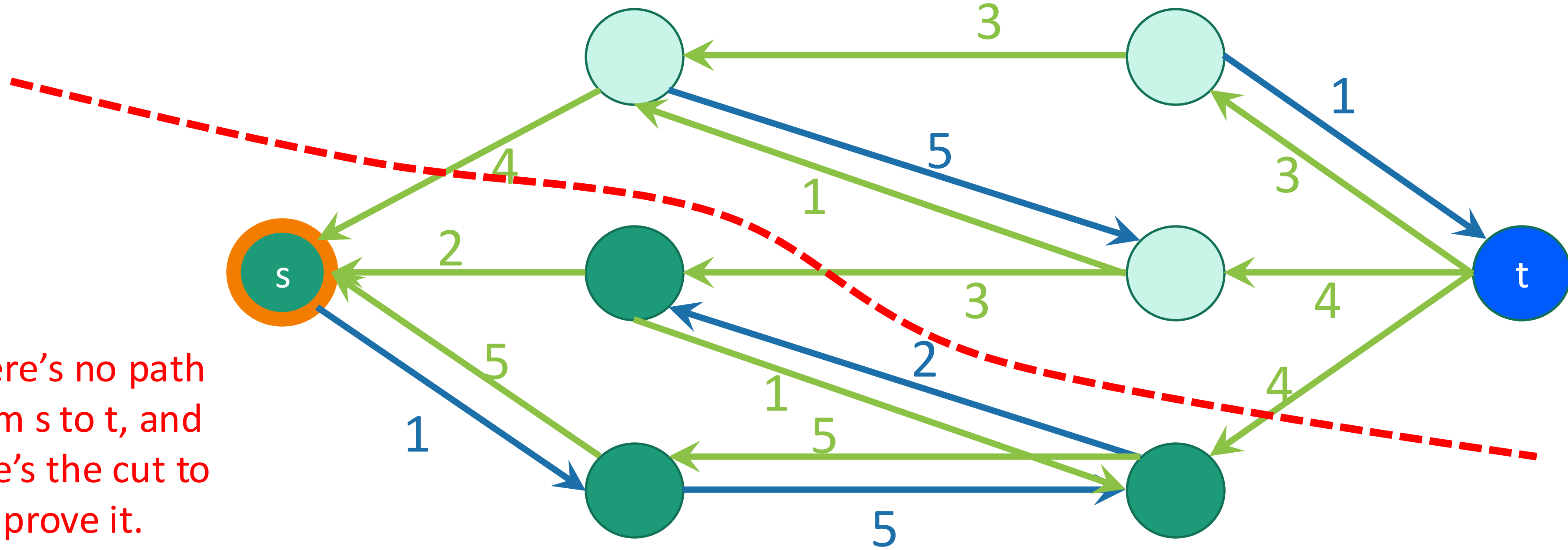


There's no path from s to t, and here's the cut to prove it.

Aside

- How do we find this cut?
- If we run BFS to find a path from s to t, and we can't find one, then the cut is

{stuff BFS can reach}, {stuff BFS can't reach}

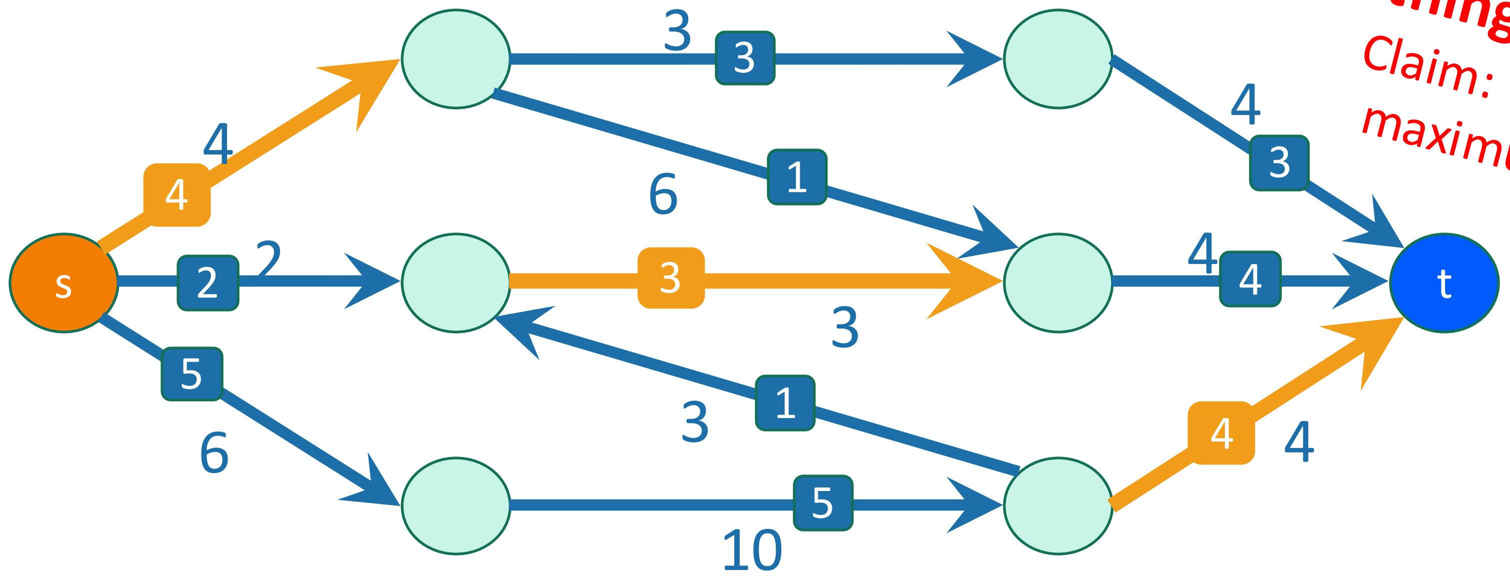


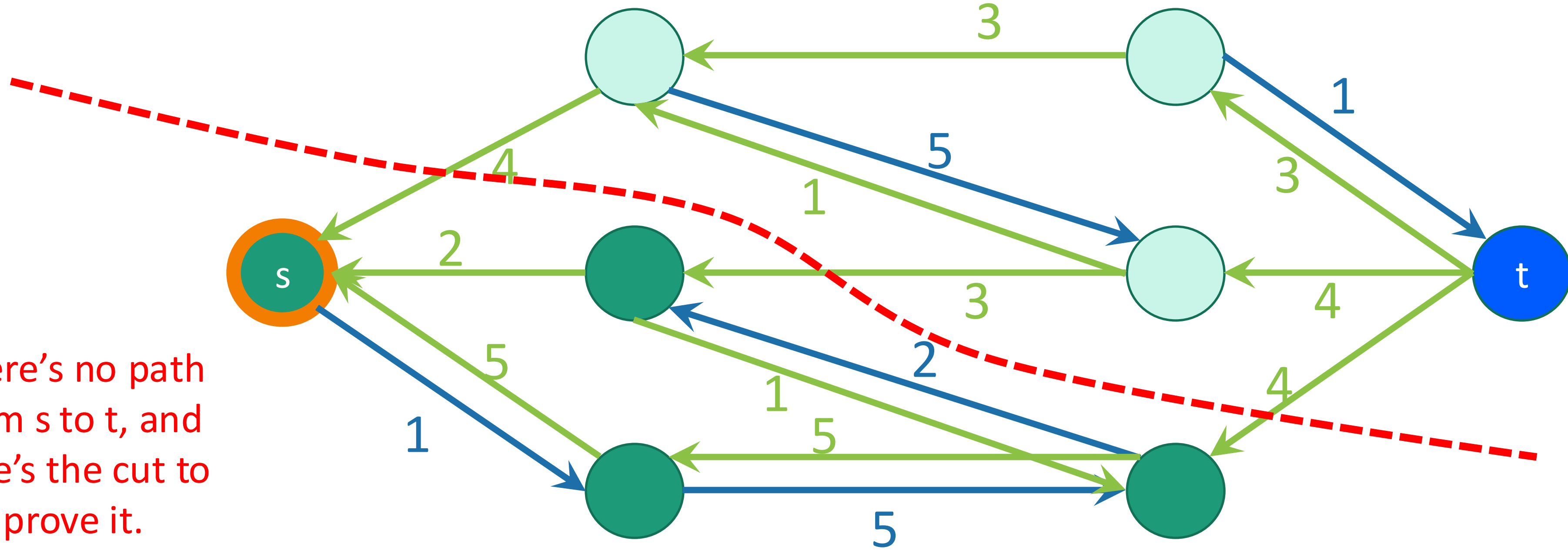
There's no path from s to t, and here's the cut to prove it.

Example of Ford-Fulkerson

Now we have nothing left to do!

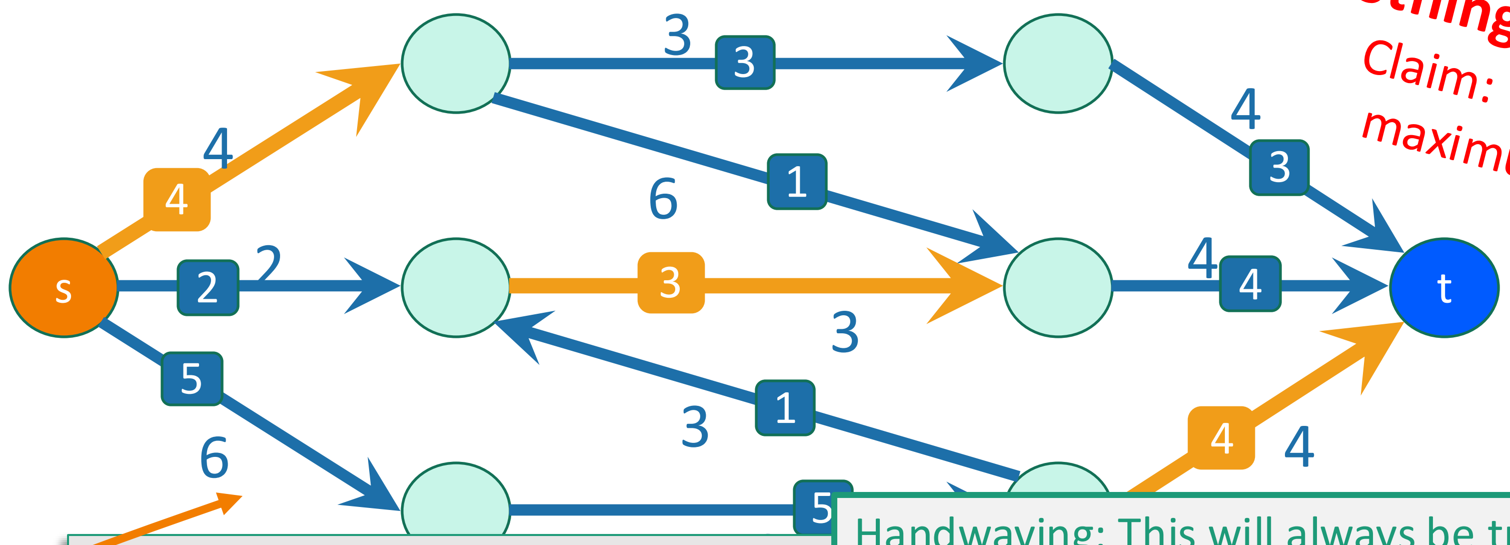
Claim: This is the maximum flow.





There's no path from s to t, and here's the cut to prove it.

Example of Ford-Fulkerson



Now we have nothing left to do!

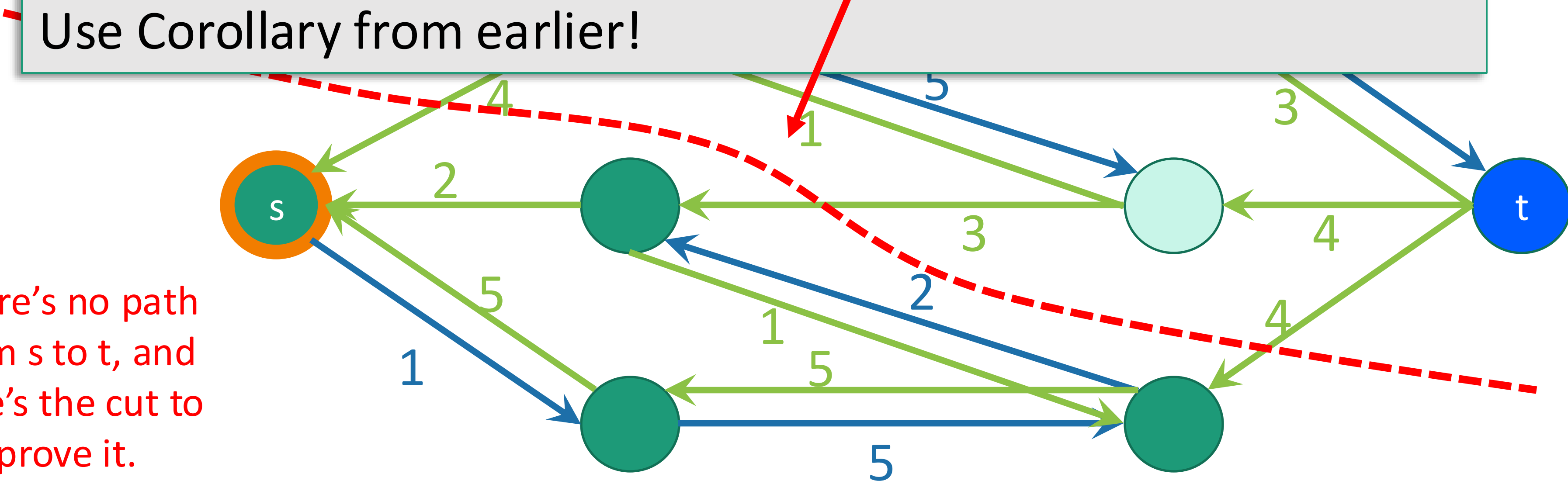
Claim: This is the maximum flow.

Handwaving: This will always be true!

Why is this a max flow?

this flow value $=$ **this cut** cost

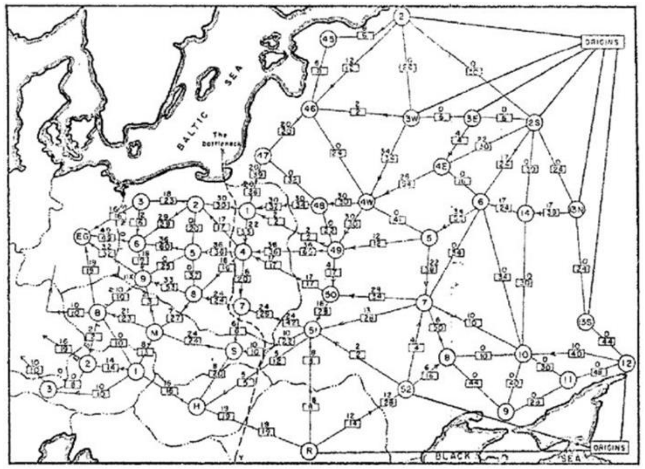
Use Corollary from earlier!



There's no path from s to t, and here's the cut to prove it.

What have we learned?

- Max s-t flow is equal to min s-t cut!
  - The USSR and the USA were trying to solve the same problem…
- Useful corollary:
  - To certify that you have a max flow, it's enough to find a cut with the same cost.
  - To certify that you have a min cut, it's enough to find a flow with the same value.
- The Ford-Fulkerson algorithm can find the min-cut/max-flow.
  - Repeatedly improve your flow along an augmenting path.



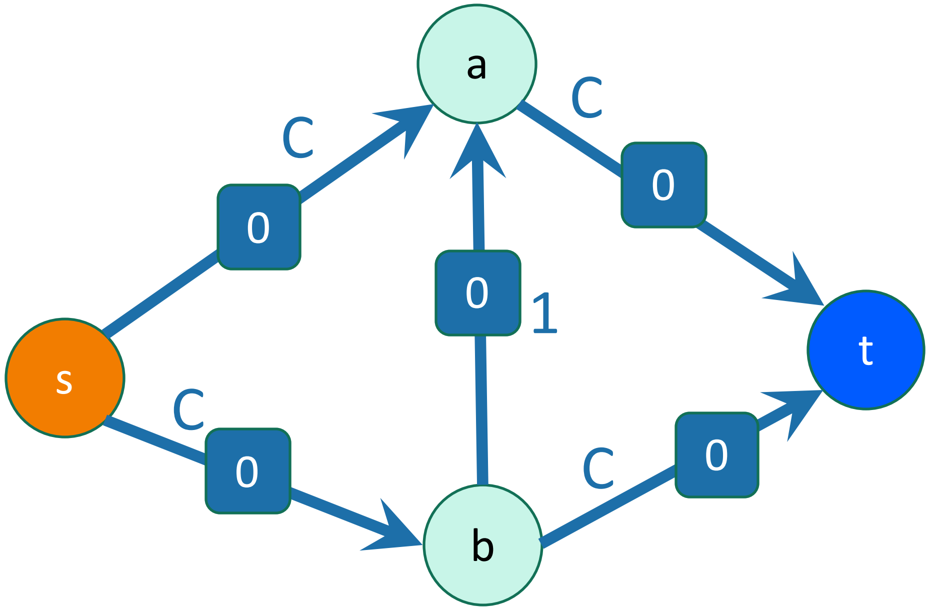
Our usual questions
about Ford-Fulkerson

- Does it work?
  - Yep, just showed that (or, hand-waved at it)

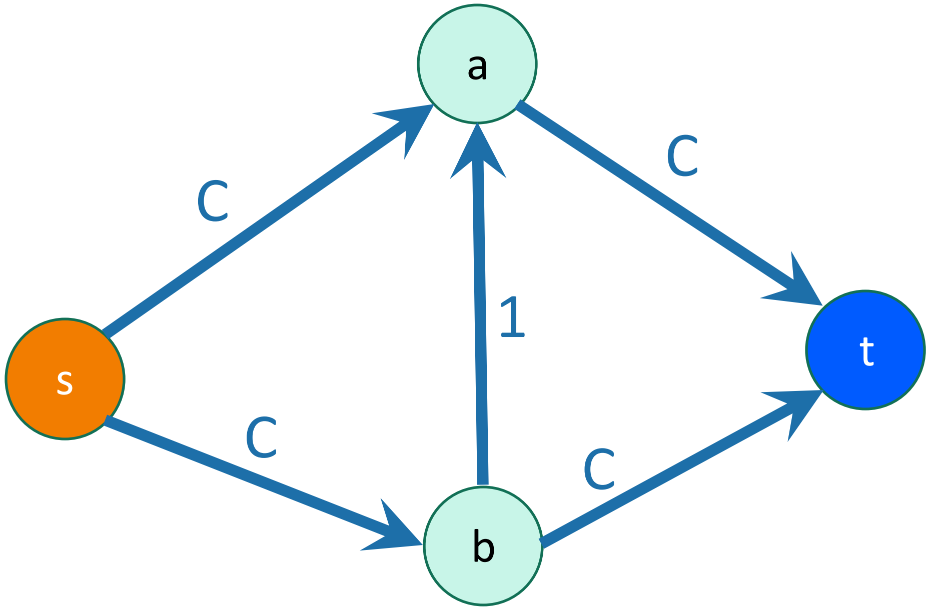
- Is it fast?
  - Depends on how we pick the augmenting paths!

Why should we be concerned?

Suppose we just picked paths arbitrarily.



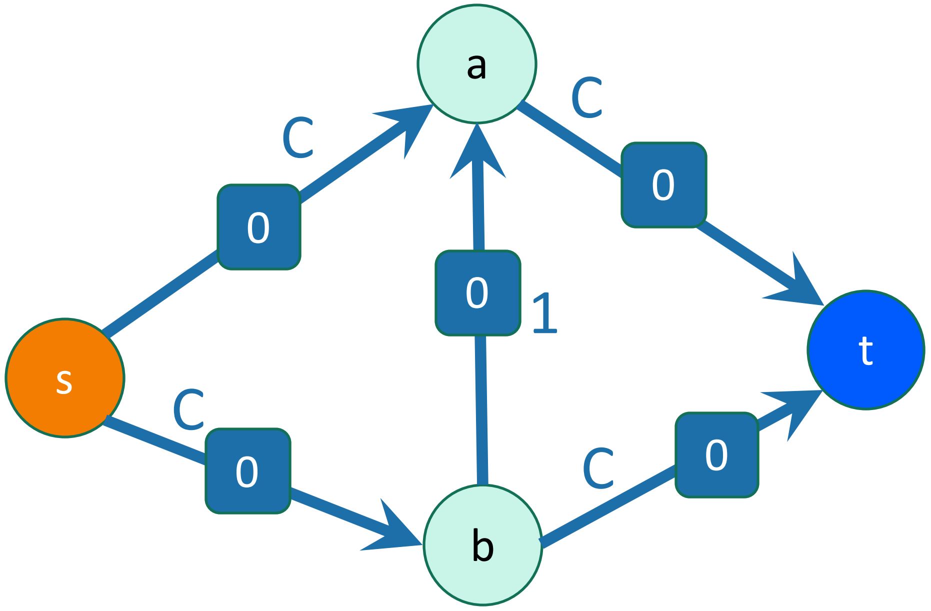
Choose a really big number C.

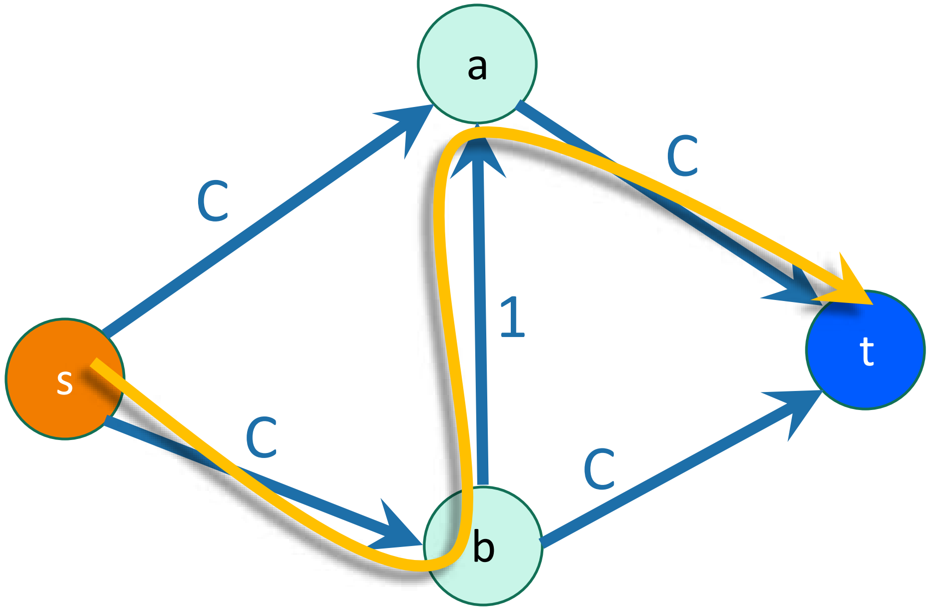


Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.

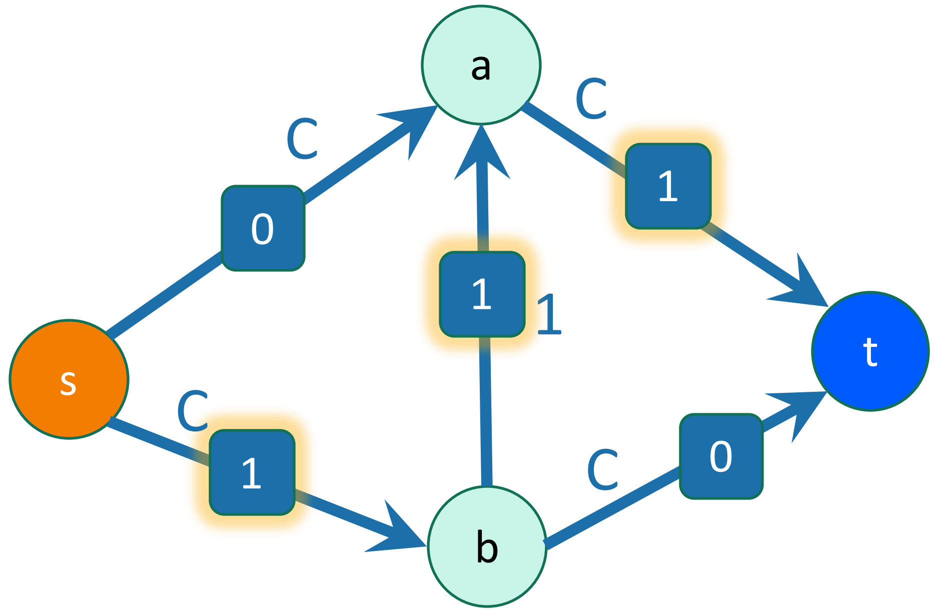


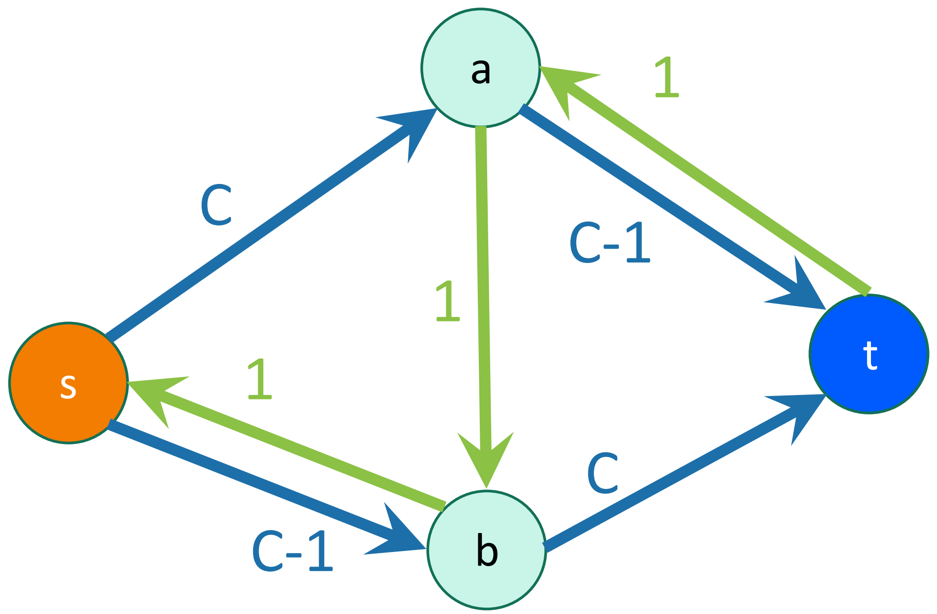


Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.



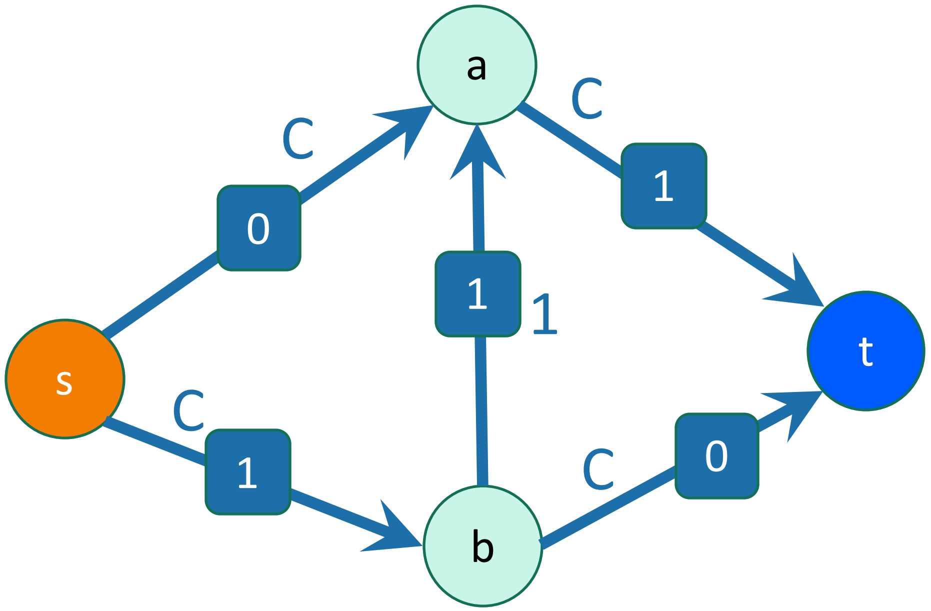


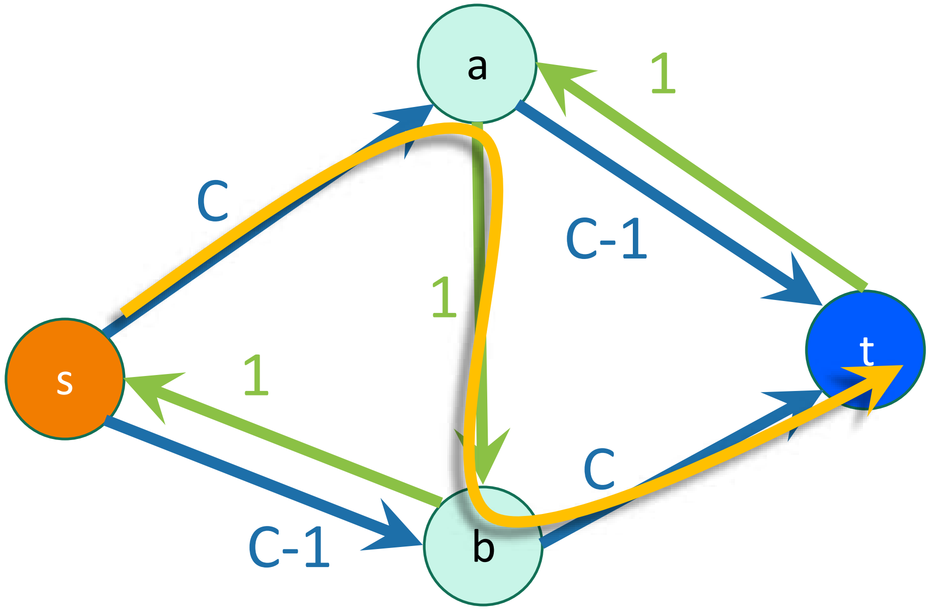
The edge (b,a) disappeared from the residual graph!

Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.

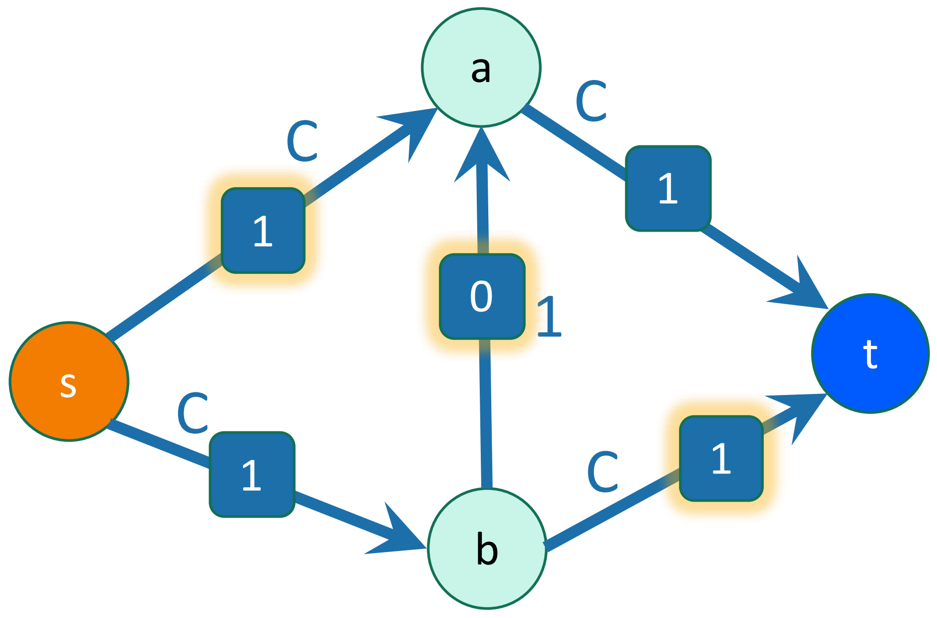


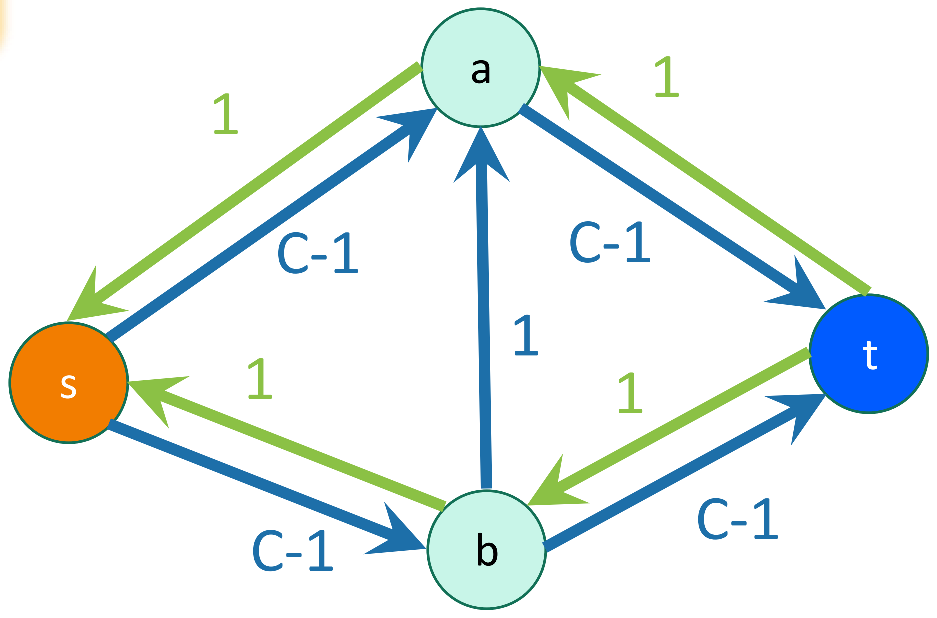


Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.



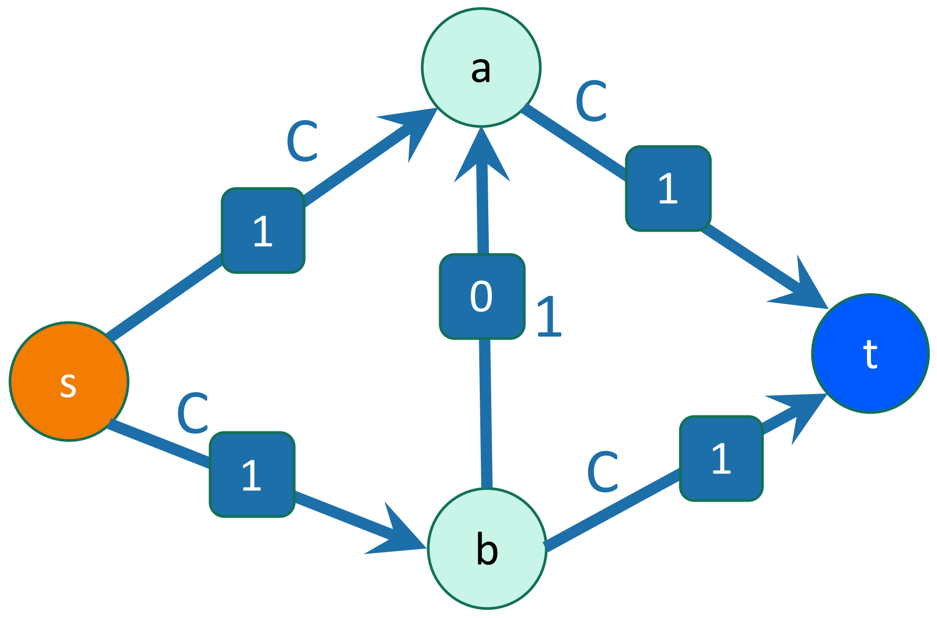


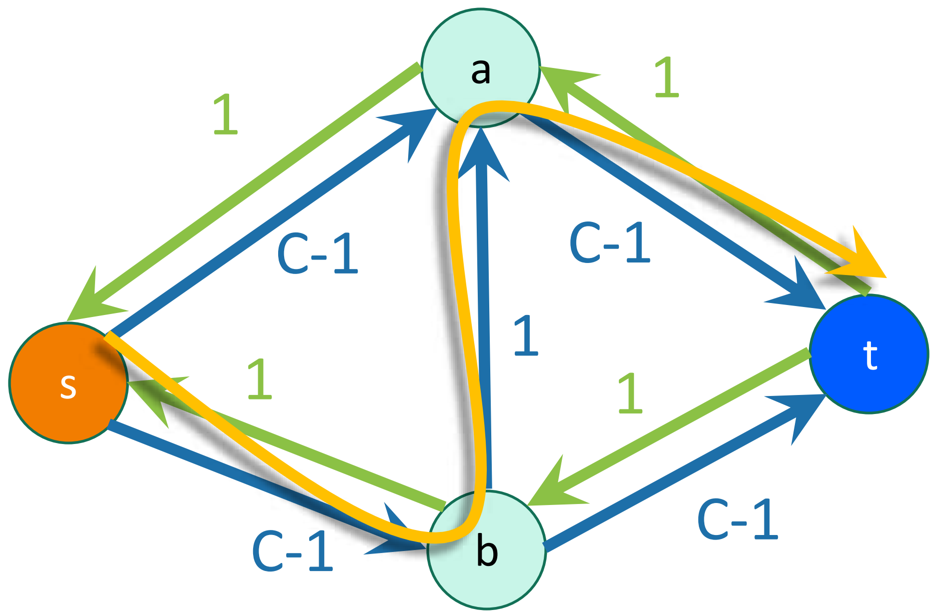
The edge (b,a) re-appeared in the residual graph!

Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.

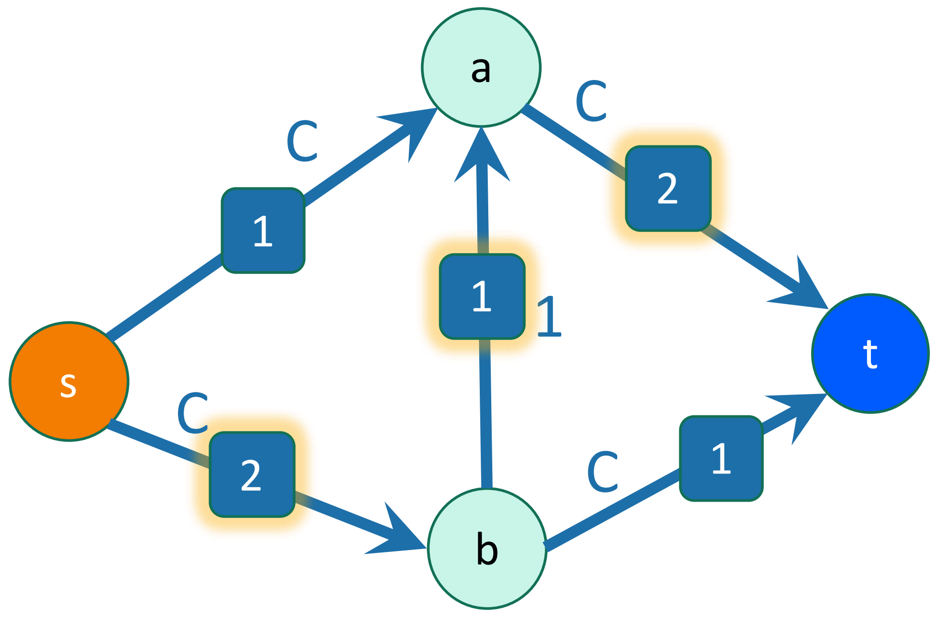


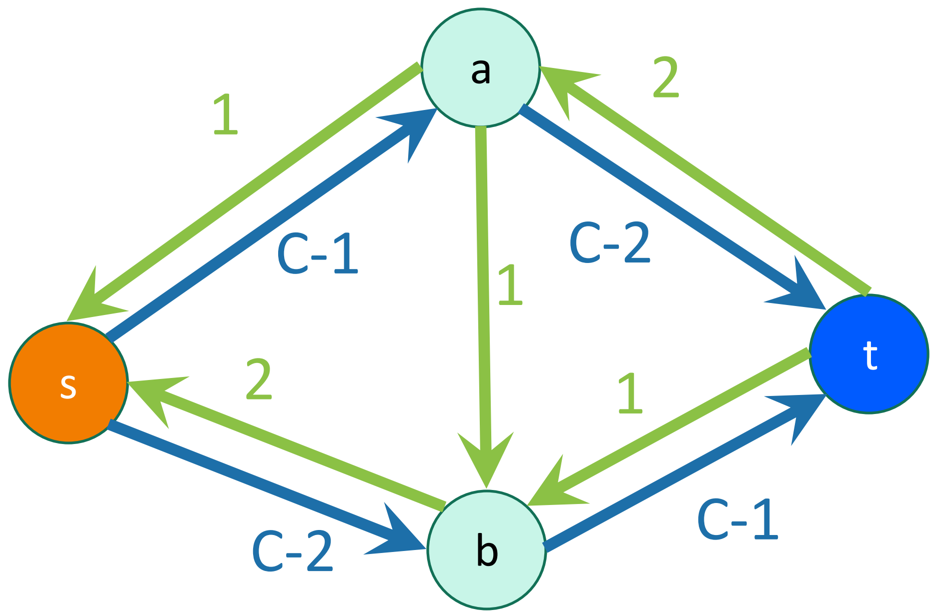


Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.



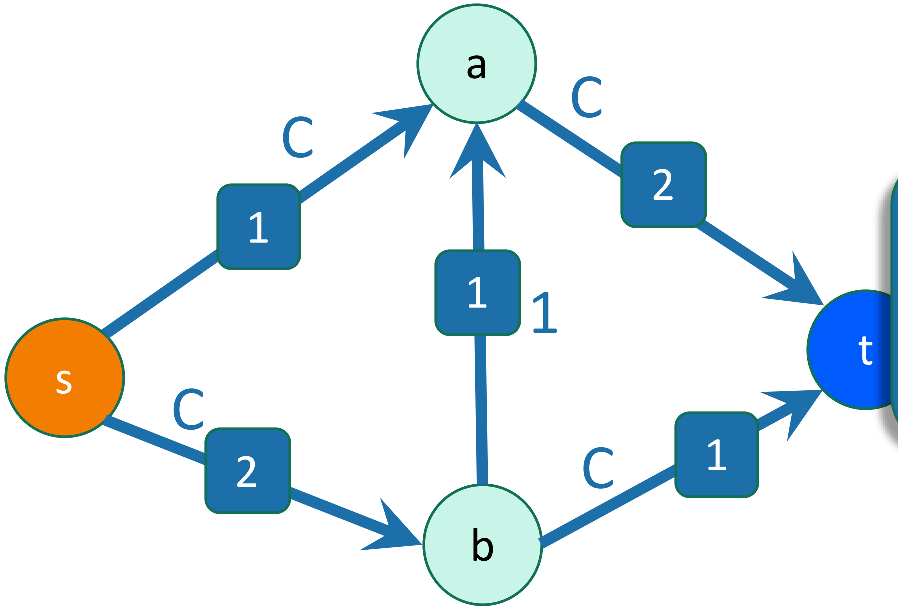


The edge (b,a) disappeared from the residual graph!

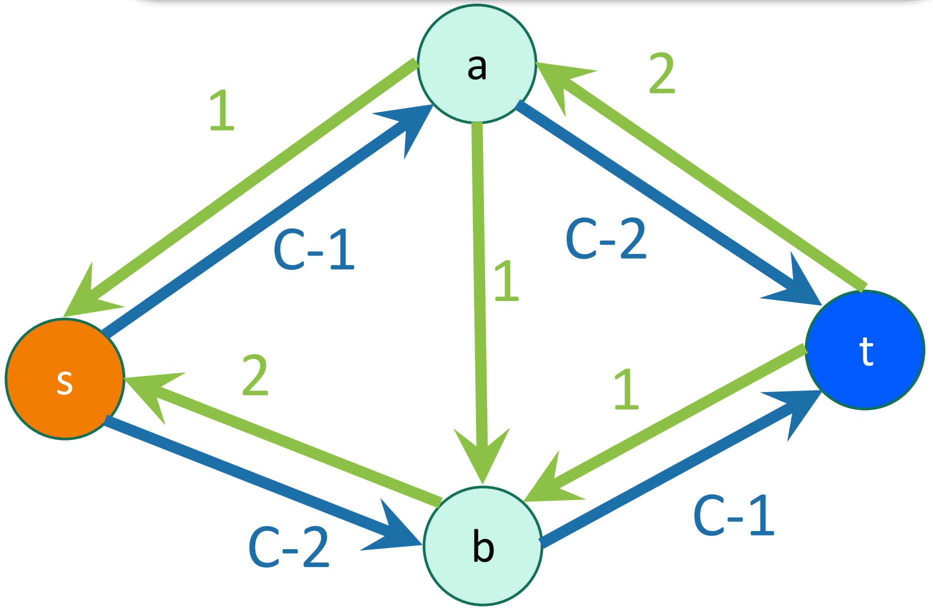
Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.



This will go on for C steps, adding flow along (b,a) and then subtracting it again.



The edge (b,a) disappeared from the residual graph!

Edmonds-Karp Algorithm

- If we run the Ford-Fulkerson algorithm, using BFS to pick augmenting paths, it's called the **Edmonds-Karp Algorithm.**

- It turns out that this will run in time $O(nm^2)$
  - You are not responsible for the proof of this fact.
  - (But you should know the statement of it ☺).

Our usual questions
about Ford-Fulkerson

- Does it work?
  - Yep, just showed that

- Is it fast?
  - Depends on how we pick the augmenting paths!
  - If we use BFS to find augmenting paths, then running time is O(nm^2) on a graph with n vertices and m edges.

One more useful observation

- If all the capacities are integers, then the flows in any max flow are also all integers.
  - When we update flows in Ford-Fulkerson, we’re only ever adding or subtracting integers.
  - Since we started with 0 (an integer), everything stays an integer.

But wait, there’s more!

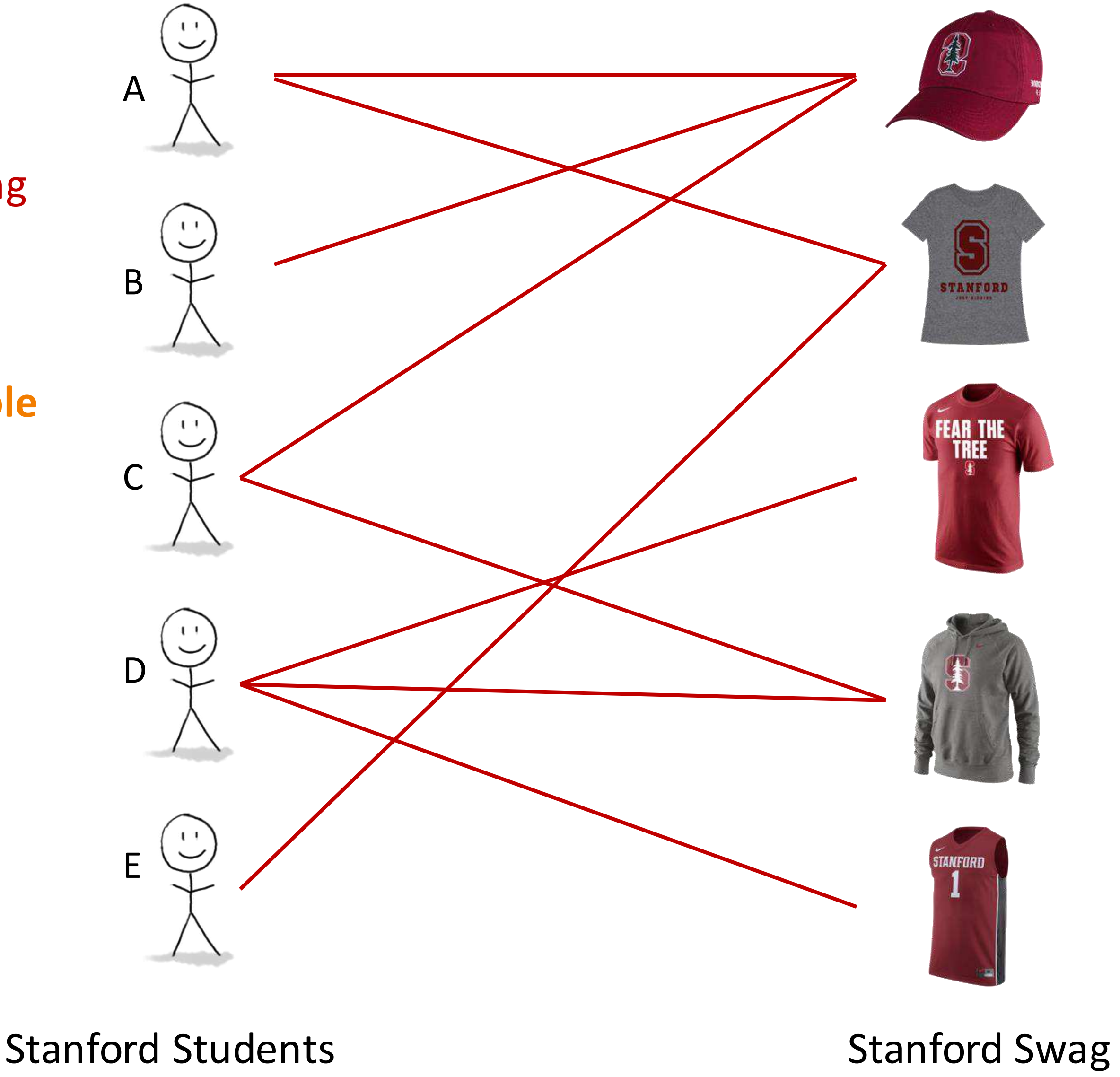
- Min-cut and max-flow are not just useful for the USA and the USSR in 1955.
- The Ford-Fulkerson algorithm is the basis for many other graph algorithms.
- For the rest of today, we’ll see a few:
  - Maximum bipartite matching
  - Integer assignment problems

For more on applications, check out this book chapter from “Algorithms” by Jeff Erickson:
https://jeffe.cs.illinois.edu/teaching/algorithms/book/11-maxflowapps.pdf

Applications!

Maximum matching in bipartite graphs

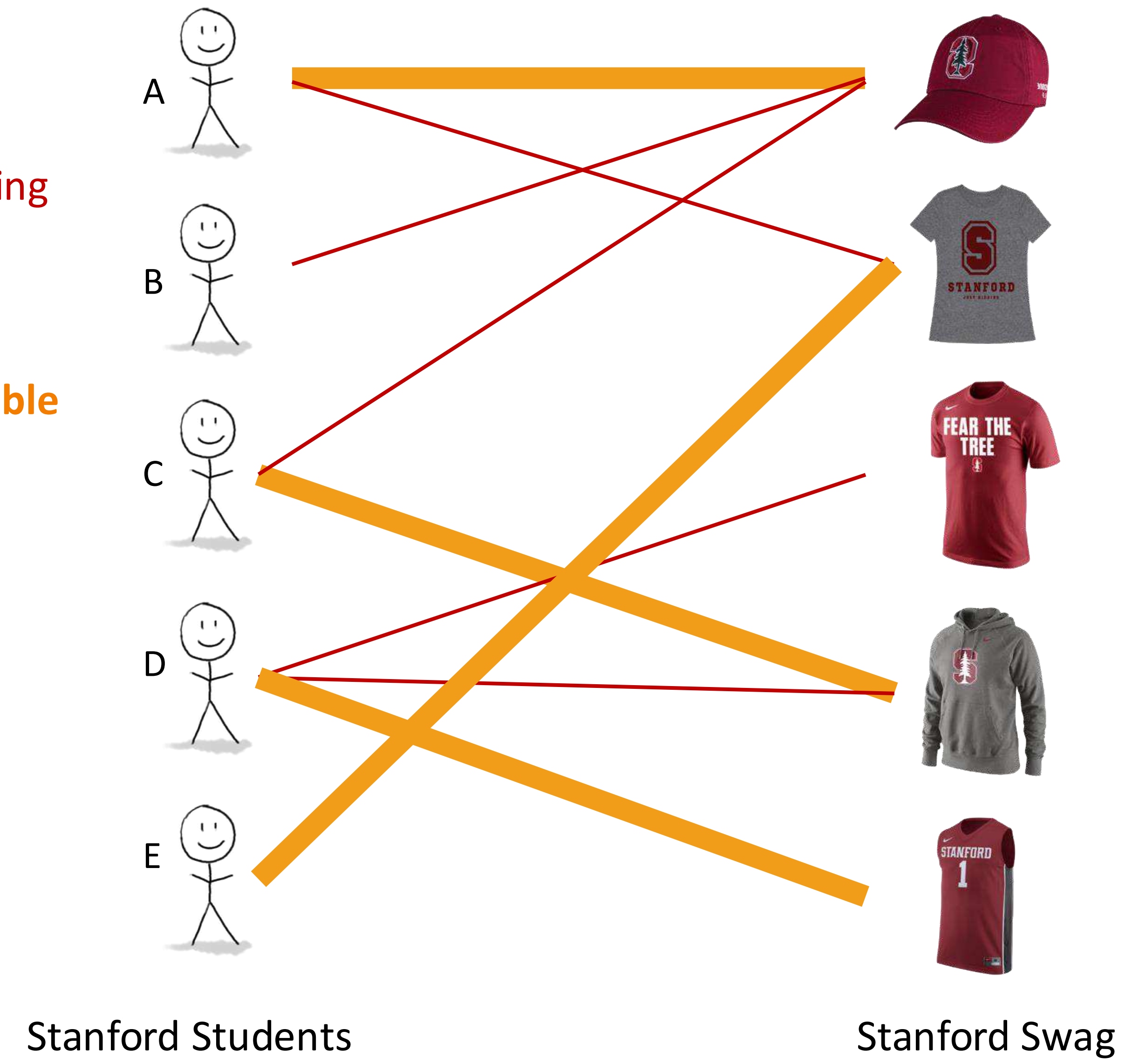
- Different students only want certain items of Stanford swag (depending on fit, style, etc).
- **How can we make as many students as possible happy?**



Maximum matching in bipartite graphs

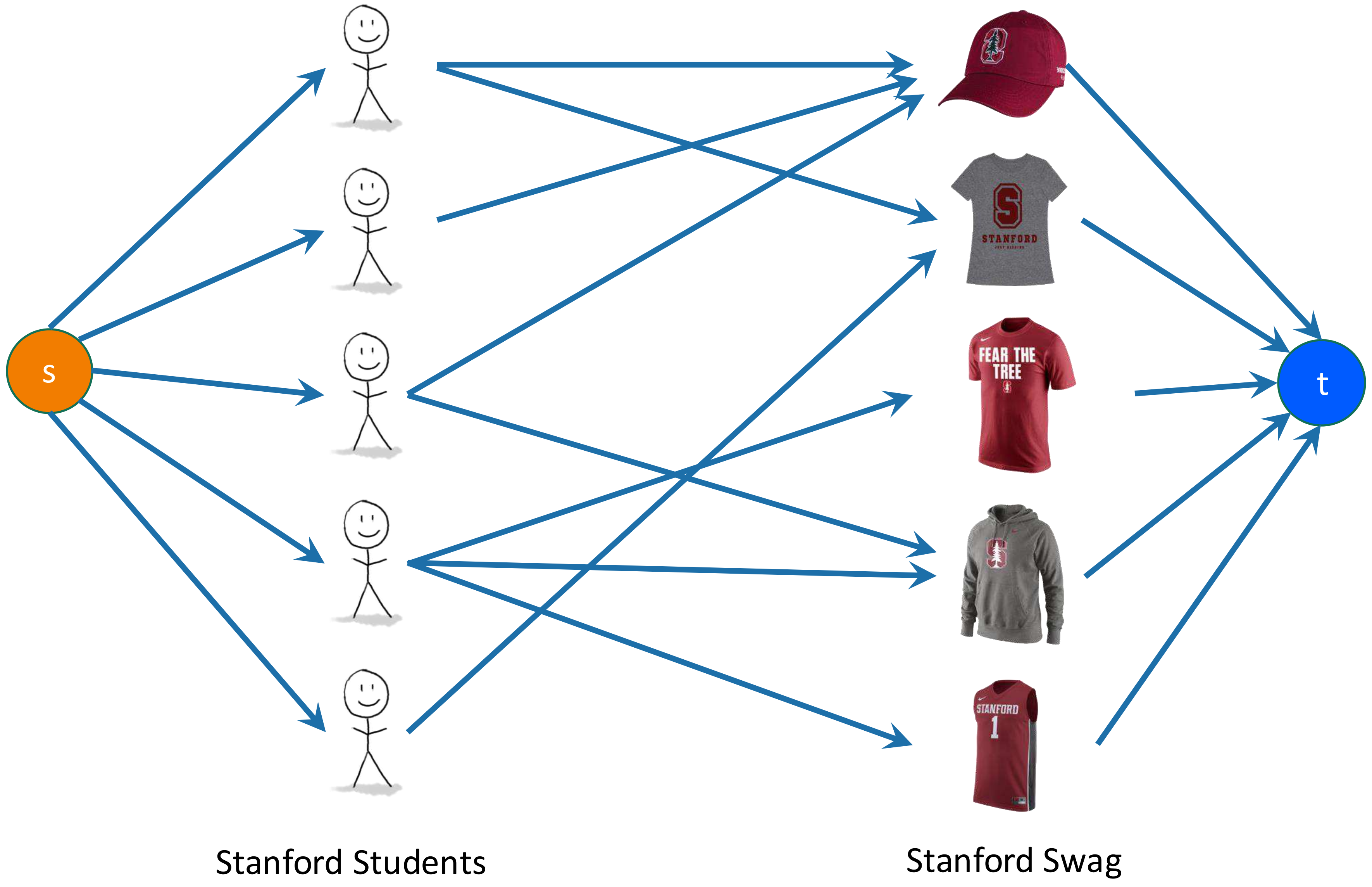
- Different students only want certain items of Stanford swag (depending on fit, style, etc).

- **How can we make as many students as possible happy?**



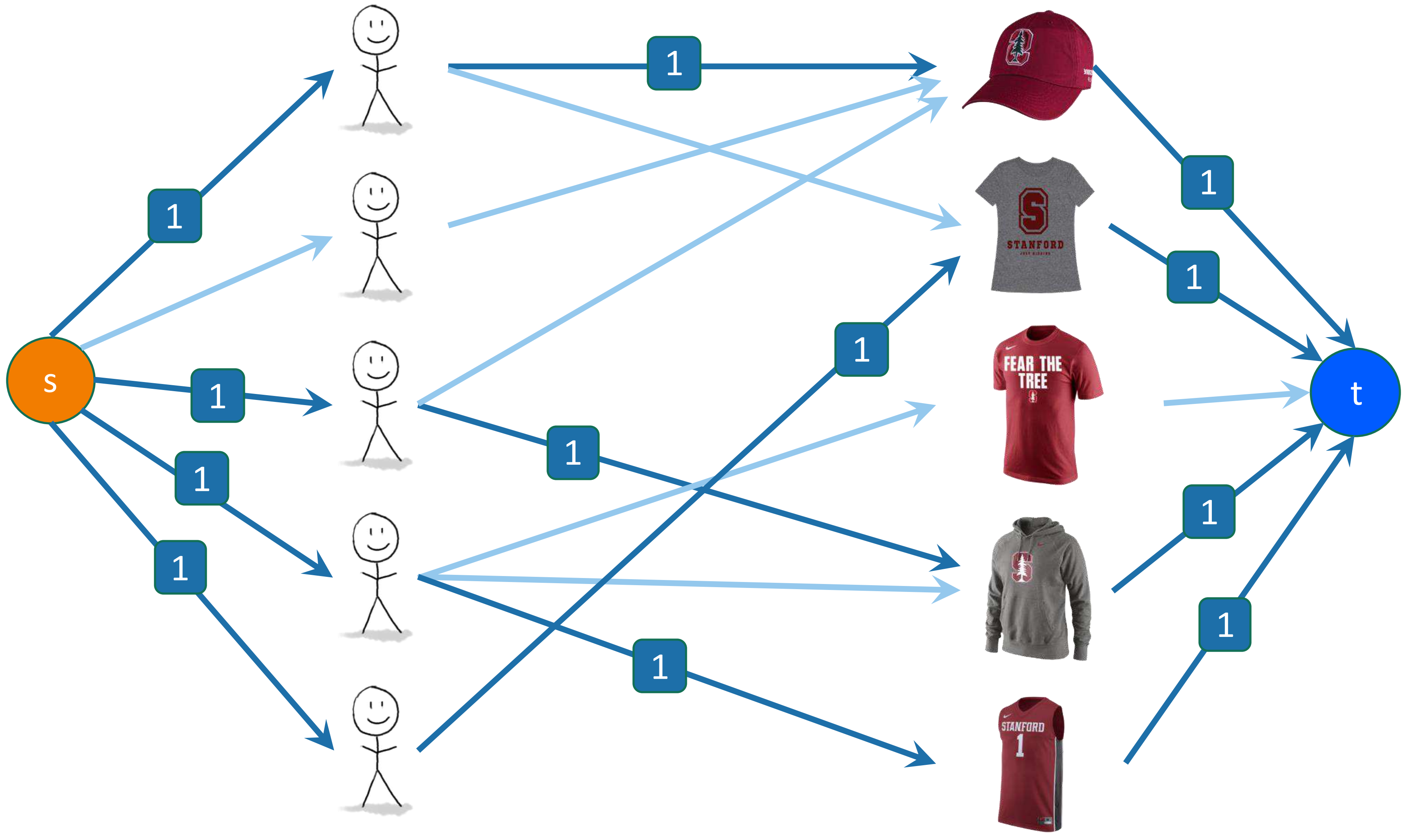
Solution via max flow

All edges have capacity 1.



Solution via max flow

All edges have capacity 1.



Stanford Students

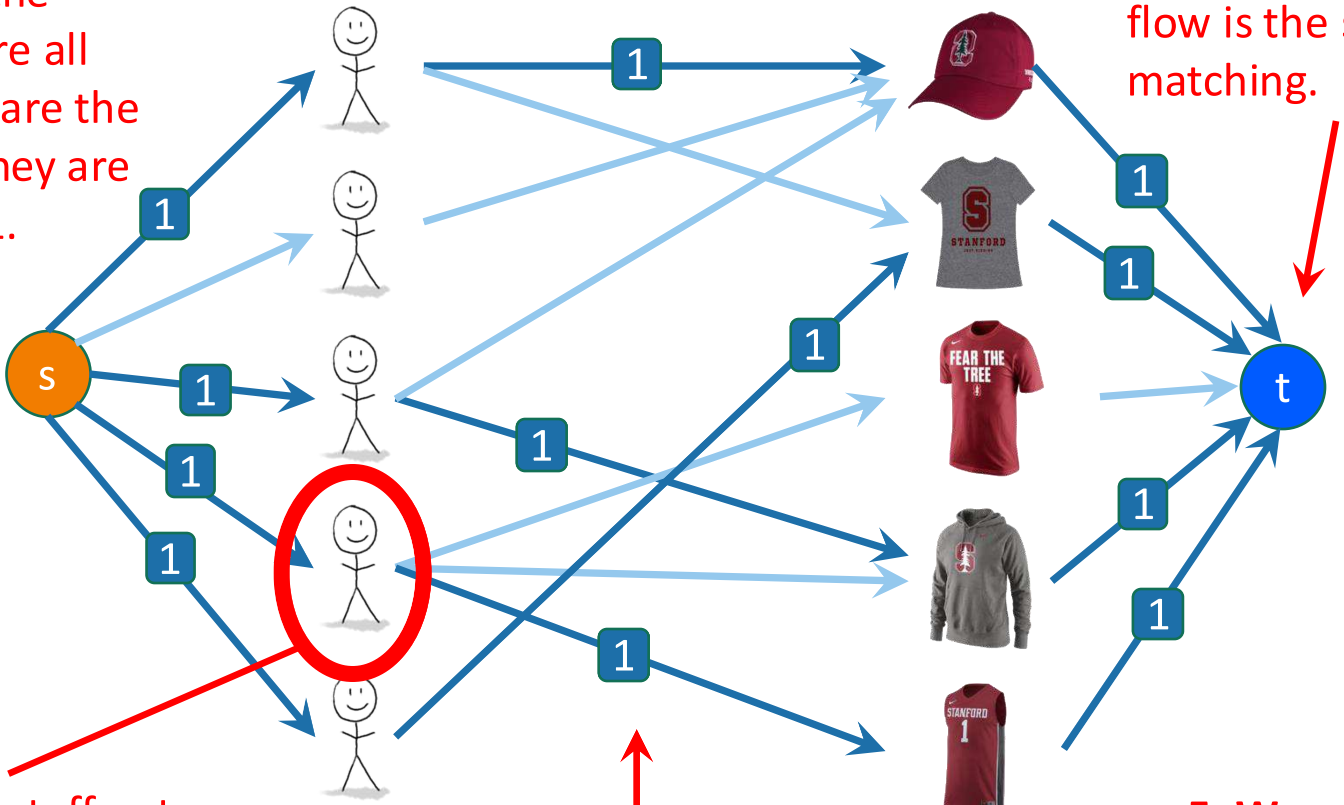
Stanford Swag

Solution via max flow

why does this work?

All edges have capacity 1.

1. Because the capacities are all integers, so are the flows – so they are either 0 or 1.



4. The value of the flow is the size of the matching.

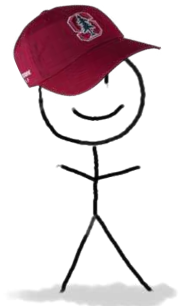
Value of this flow is 4.

2. Stuff in = stuff out means that the number of items assigned to each student 0 or 1. (And vice versa).

3. Thus, the edges with flow on them form a matching. (And, any matching gives a flow).

5. We conclude that the max flow corresponds to a max matching.

A slightly more complicated example: assignment problems



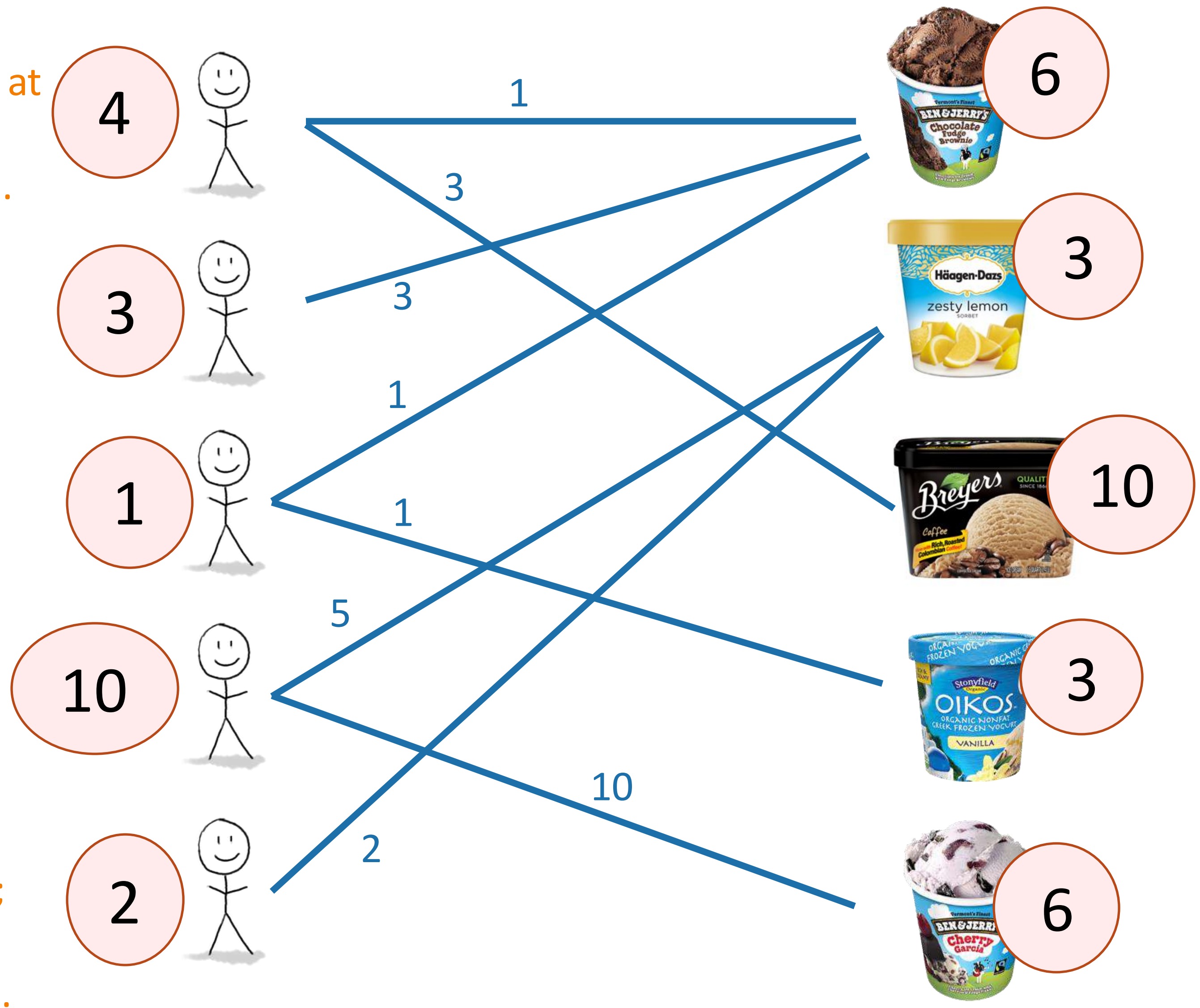
- One set X
  - Example: Stanford students
- Another set Y
  - Example: tubs of ice cream
- Each x in X can participate in c(x) matches.
  - Student x can only eat 4 scoops of ice cream.
- Each y in Y can only participate in c(y) matches.
  - Tub of ice cream y only has 10 scoops in it.
- Each pair (x,y) can only be matched c(x,y) times.
  - Student x only wants 3 scoops of flavor y
  - Student x' doesn't want any scoops of flavor y'
- **Goal: assign as many matches as possible.**

Example

How can we serve as much ice cream as possible?

This person wants 4 scoops of ice cream, at most 1 of chocolate and at most 3 coffee.

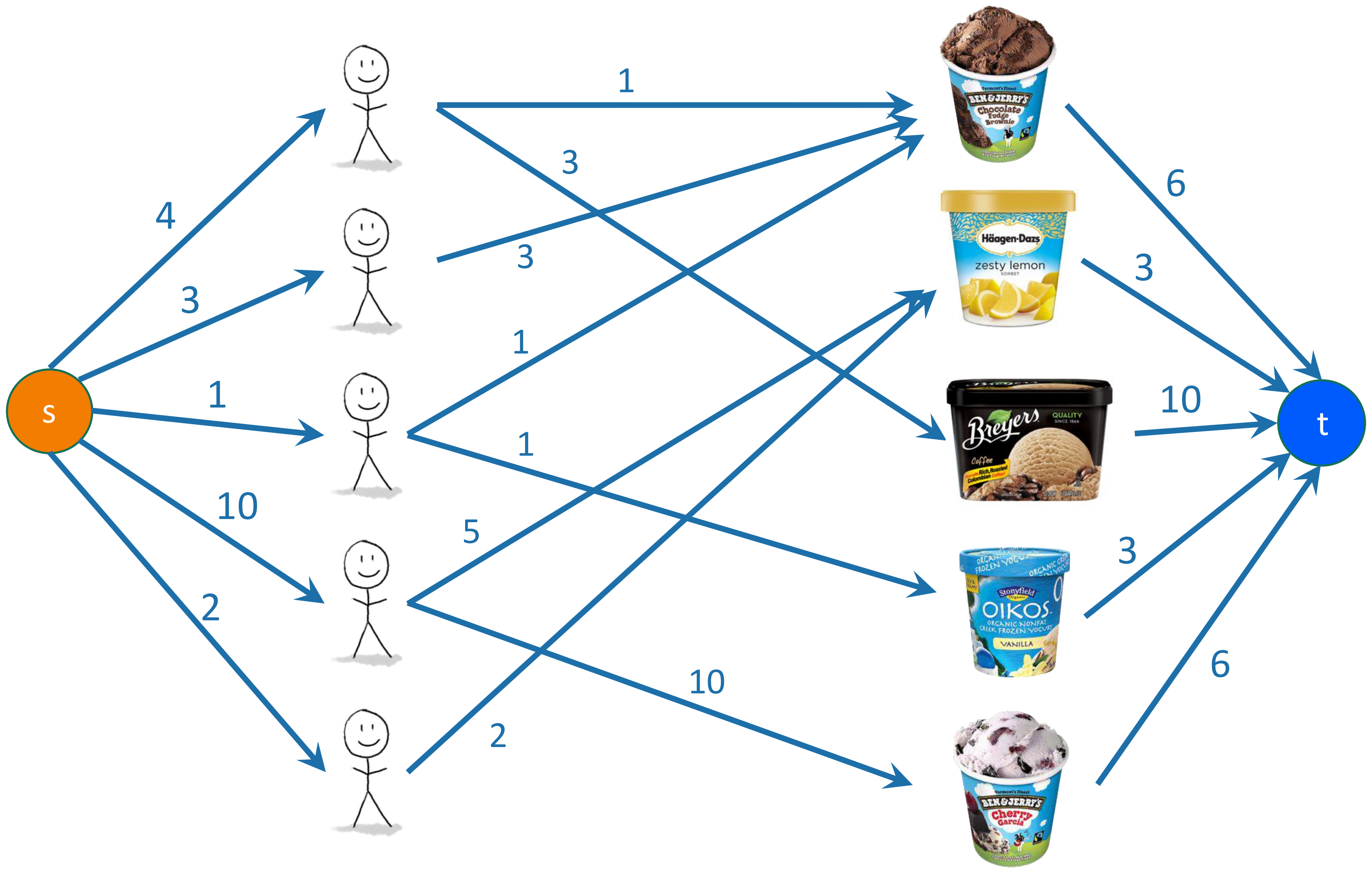
This person is vegan and not that hungry; they only want two scoops of the sorbet.


4
3
1
10
2
1
3
3
1
1
5
10
2
6
3
10
3
6
BEN & JERRY'S Chocolate Fudge Brownie
Häagen-Dazs zesty lemon sorbet
Breyers Coffee
Stonyfield OIKOS Vanilla
BEN & JERRY'S Cherry Garcia

Stanford Students

Tubs of ice cream

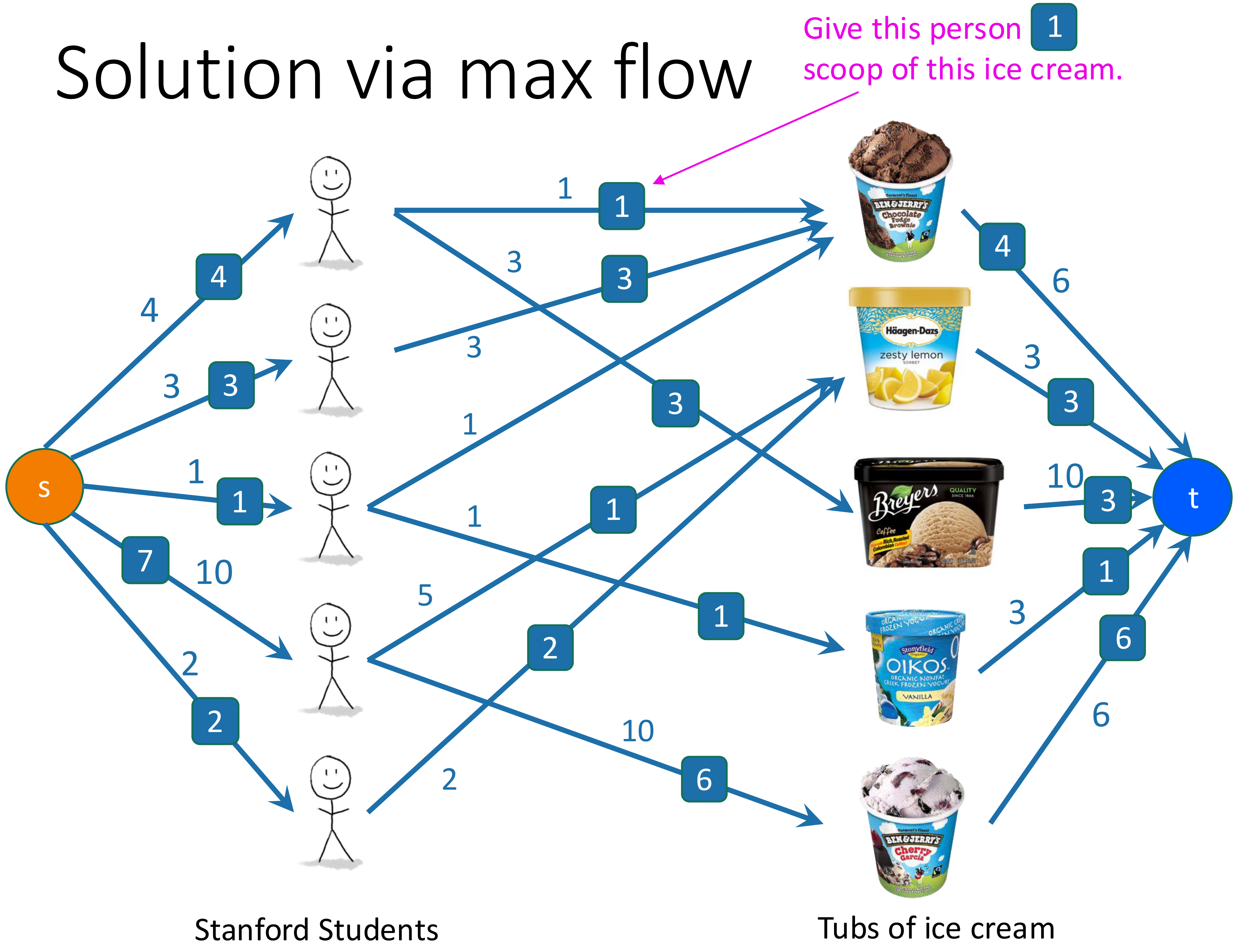
Solution via max flow



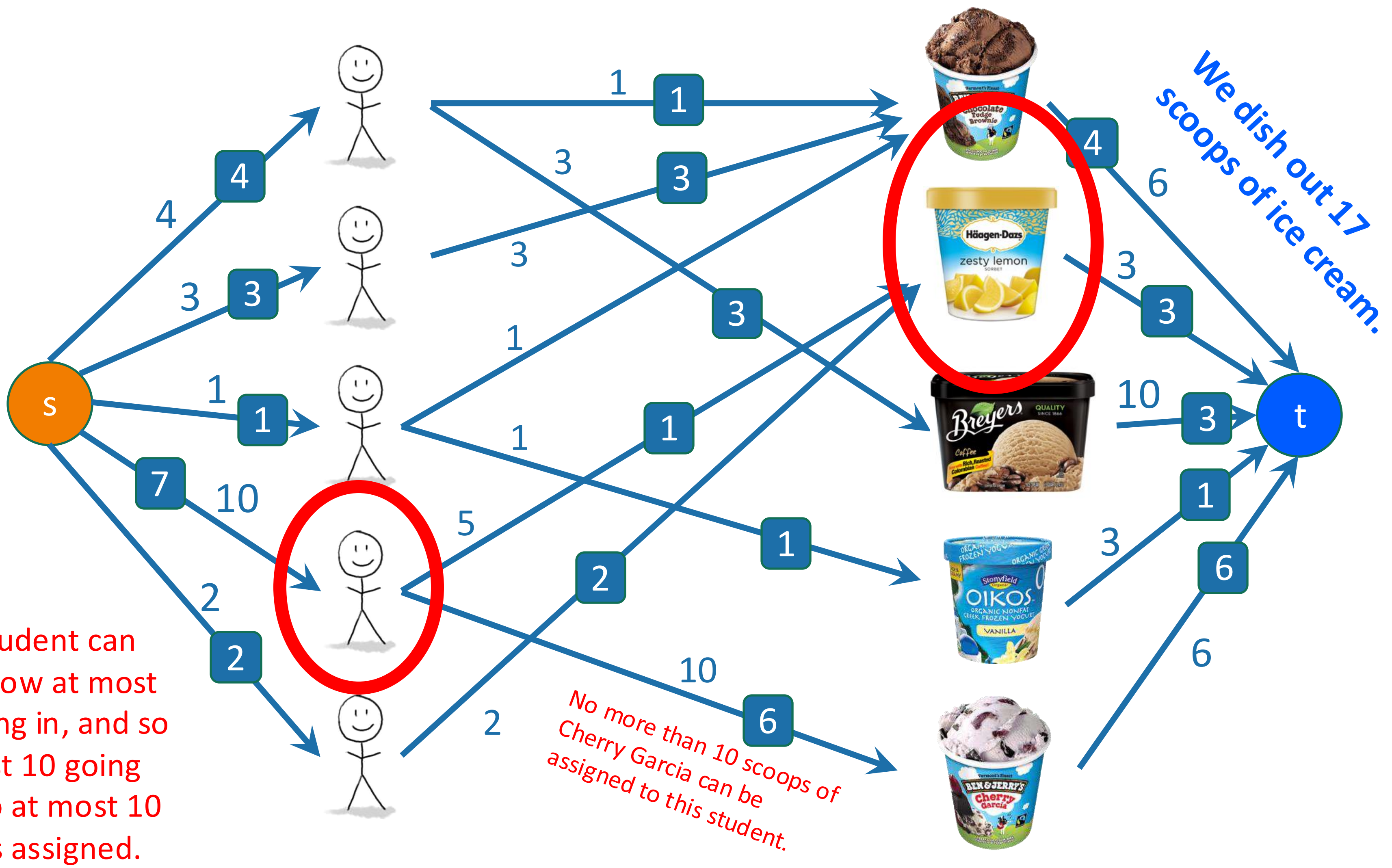
Stanford Students

Tubs of ice cream

Solution via max flow



Solution via max flow


No more than 3 scoops of sorbet can be assigned.
We dish out 17 scoops of ice cream.
s
t
4
4
3
3
1
1
7
10
2
2
1
1
3
3
3
3
1
1
1
1
5
1
2
2
10
6
4
6
3
3
10
3
3
1
6
6
No more than 10 scoops of Cherry Garcia can be assigned to this student.

This student can have flow at most 10 going in, and so at most 10 going out, so at most 10 scoops assigned.

As before, flows correspond to assignments, and max flows correspond to max assignments.

What have we learned?

- Max flows and min cuts aren't just for railway routing.
  - Immediately, they apply to other sorts of routing too!
  - But also they are useful for assigning items to Stanford students!

Recap

- Today we talked about s-t cuts and s-t flows.
- The **Min-Cut Max-Flow Theorem** says that minimizing the cost of cuts is the same as maximizing the value of flows.
- The Ford-Fulkerson algorithm does this!
  - Find an augmenting path
  - Increase the flow along that path
  - Repeat until you can’t find any more paths and then you’re done!
- An important algorithmic primitive!
  - eg, assignment problems.

Next time

- What we’ve done, and what’s to come!
  - No new material (that will be on the exam)
  - Some recap
  - Some hints of what’s to come if you keep taking algorithms classes!