Module 11 – Network-based Algorithms

SIT320 – Advanced Algorithms

Dr. Nayyar Zaidi

Today

- Minimum Cuts!
    - Karger's algorithm

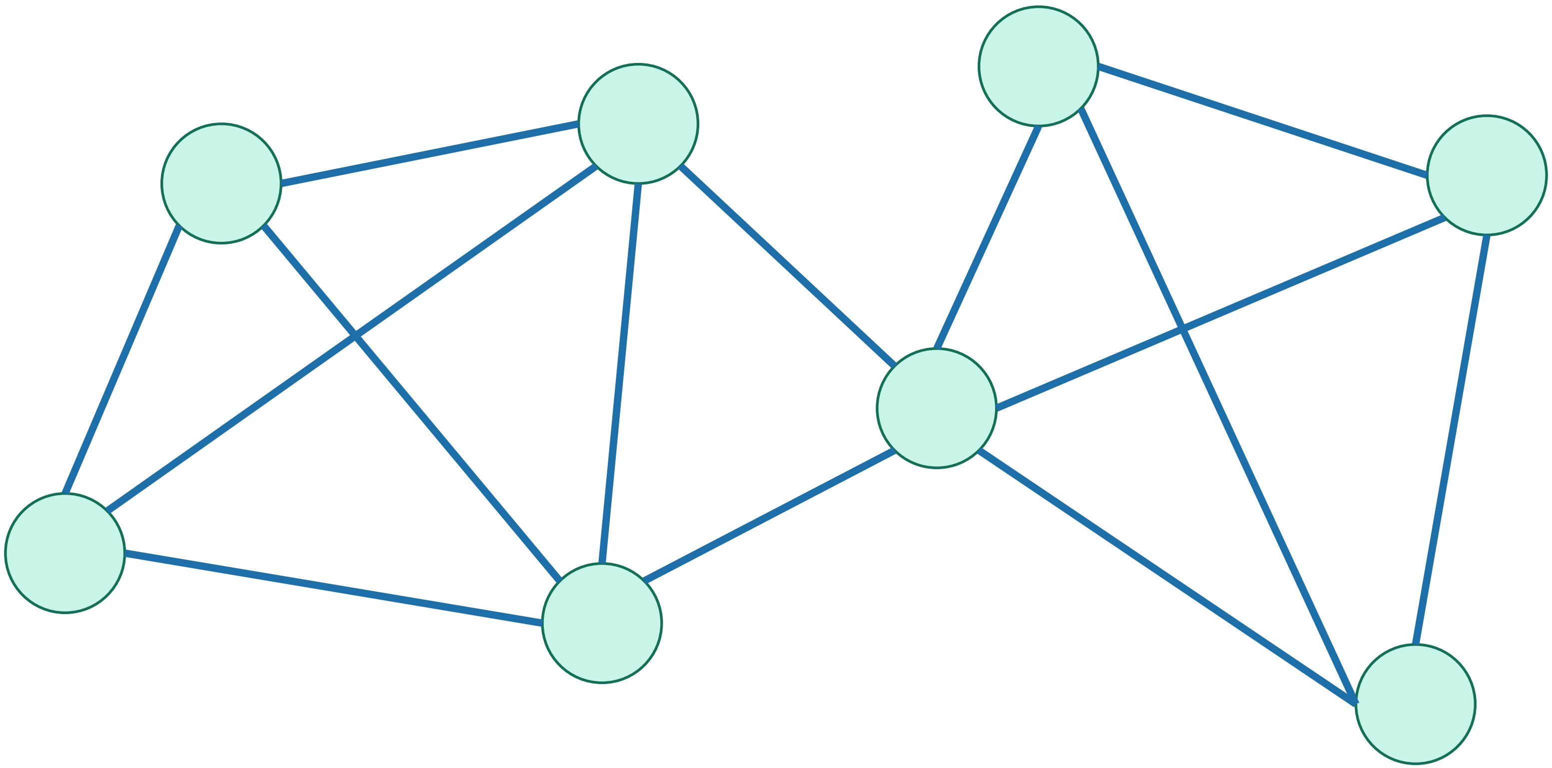
- Minimum s-t cuts
- Maximum s-t flows

- The Ford-Fulkerson Algorithm
    - Finds min cuts and max flows!

Recall: cuts in graphs

*For today, all graphs are **undirected and unweighted.**

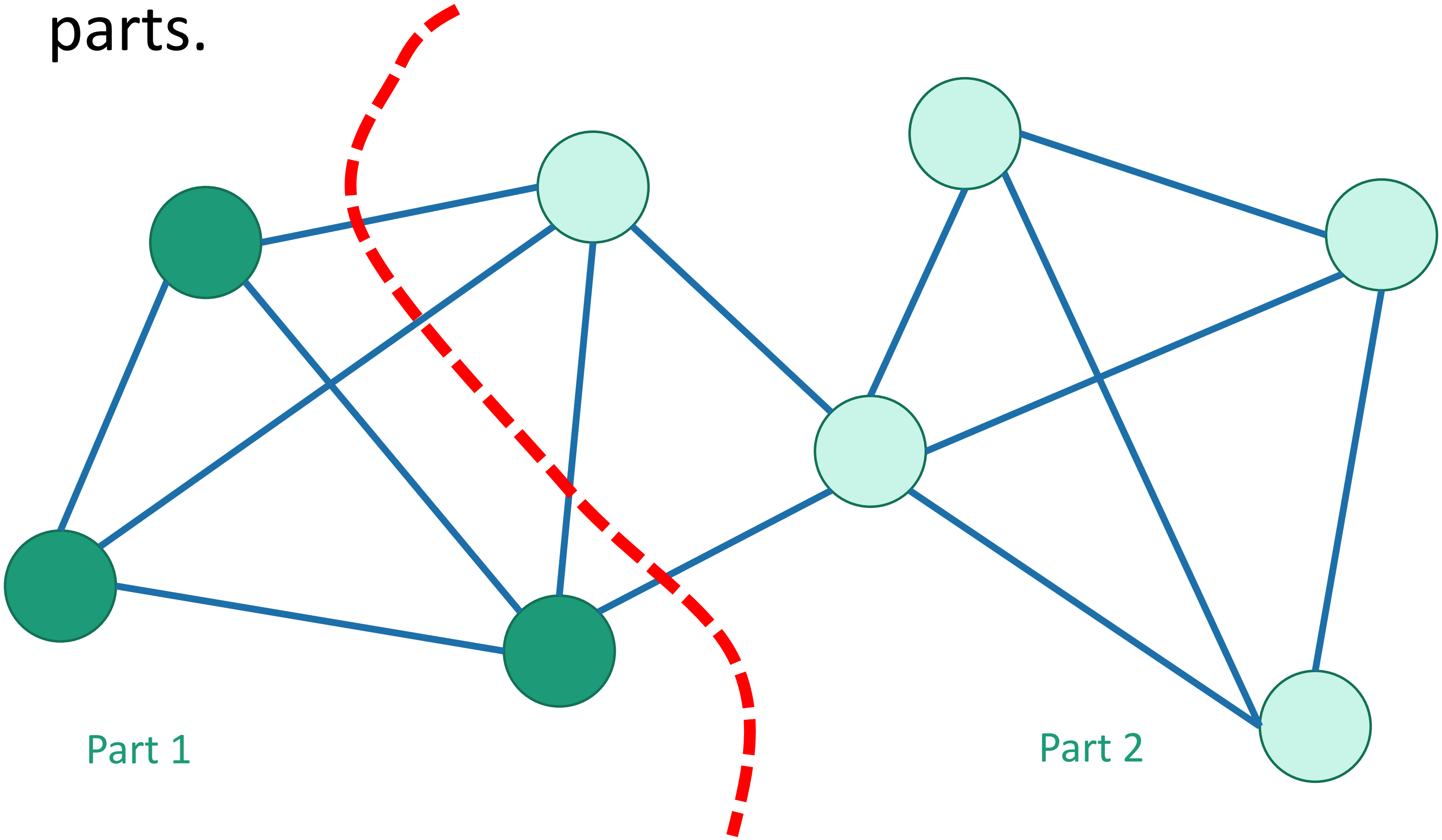
- A cut is a partition of the vertices into two nonempty parts.



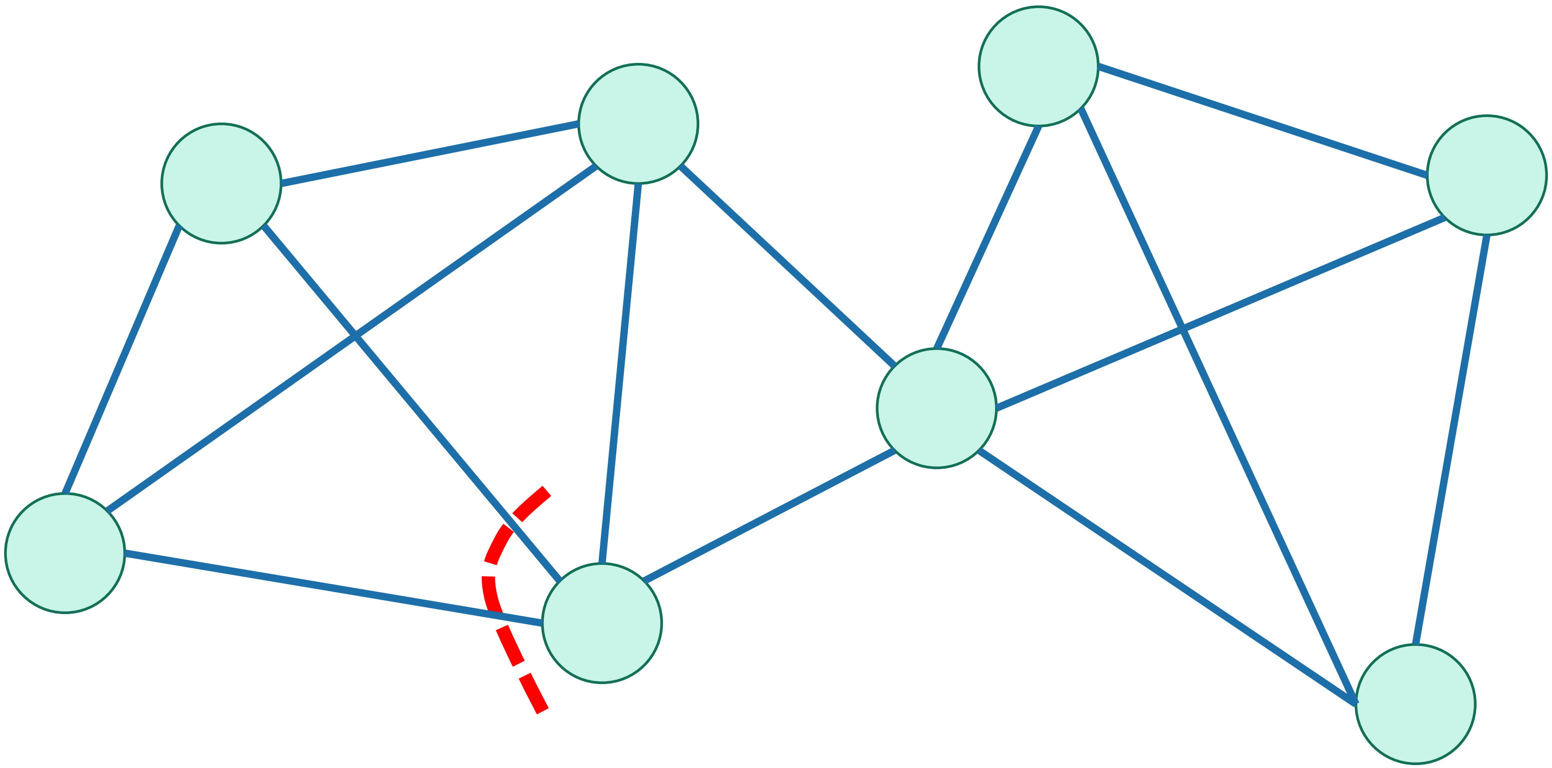
Recall: cuts in graphs

*For today, all graphs are **undirected and unweighted.**

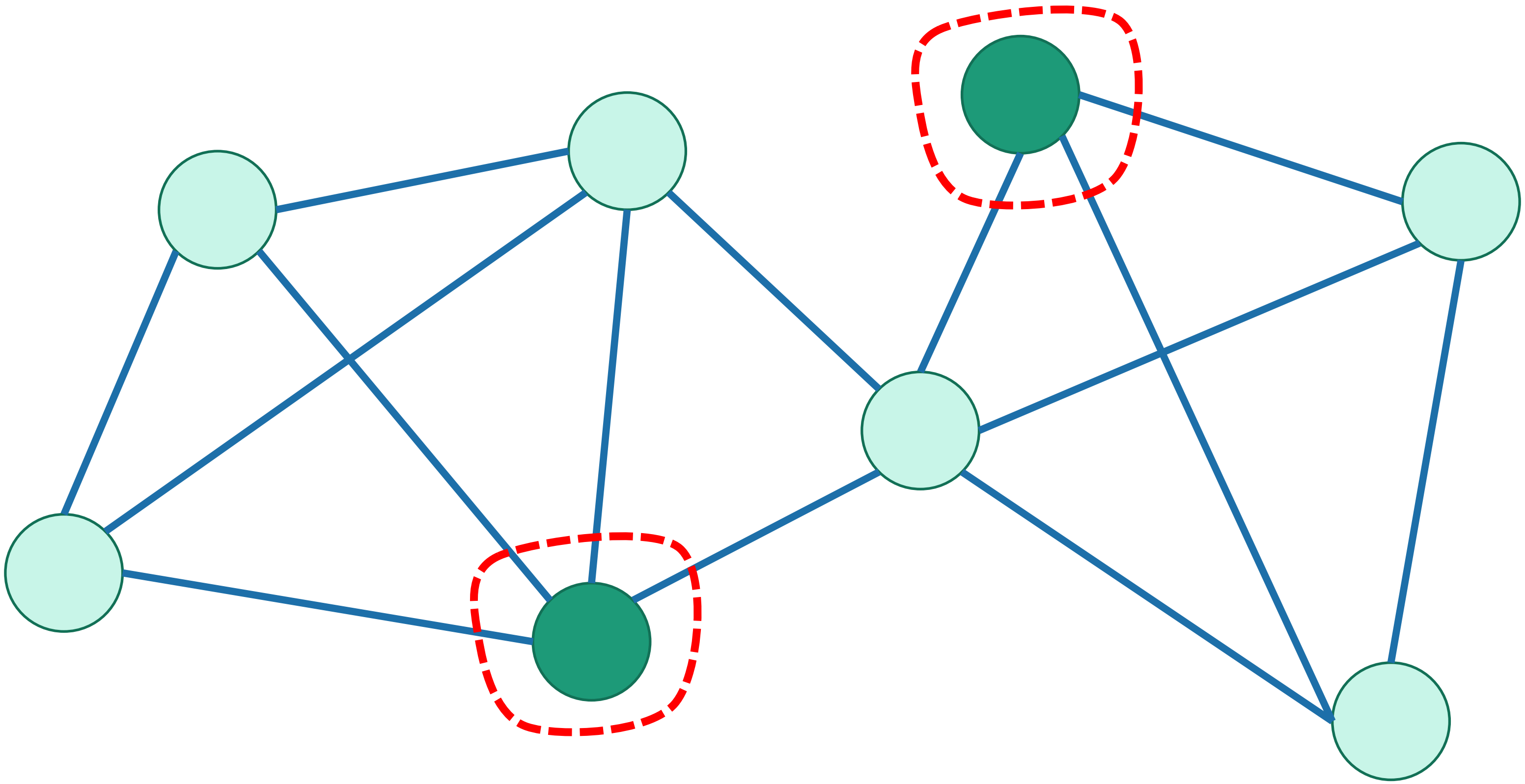
- A cut is a partition of the vertices into two nonempty parts.



This is not a cut



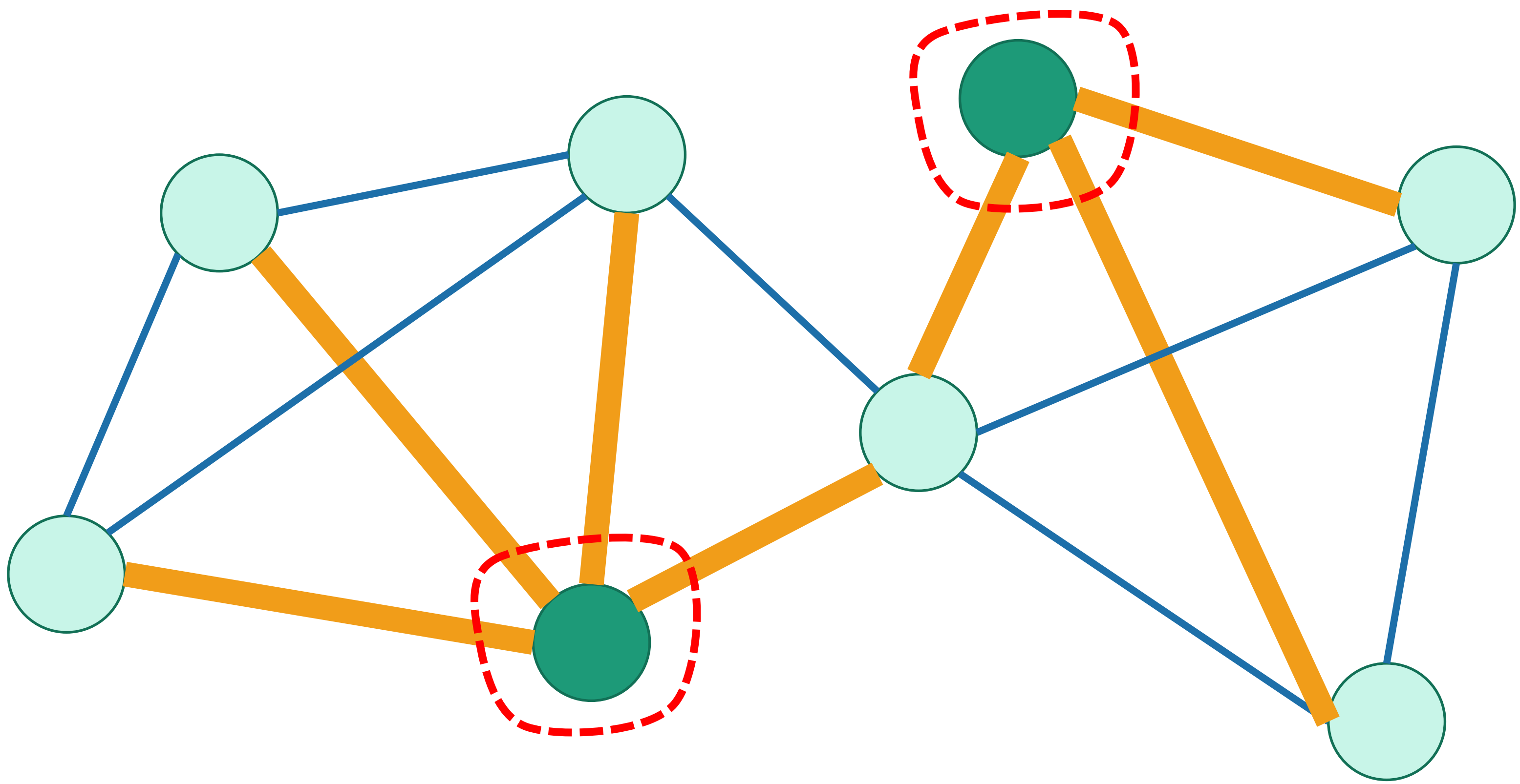
This is a cut



This is a cut

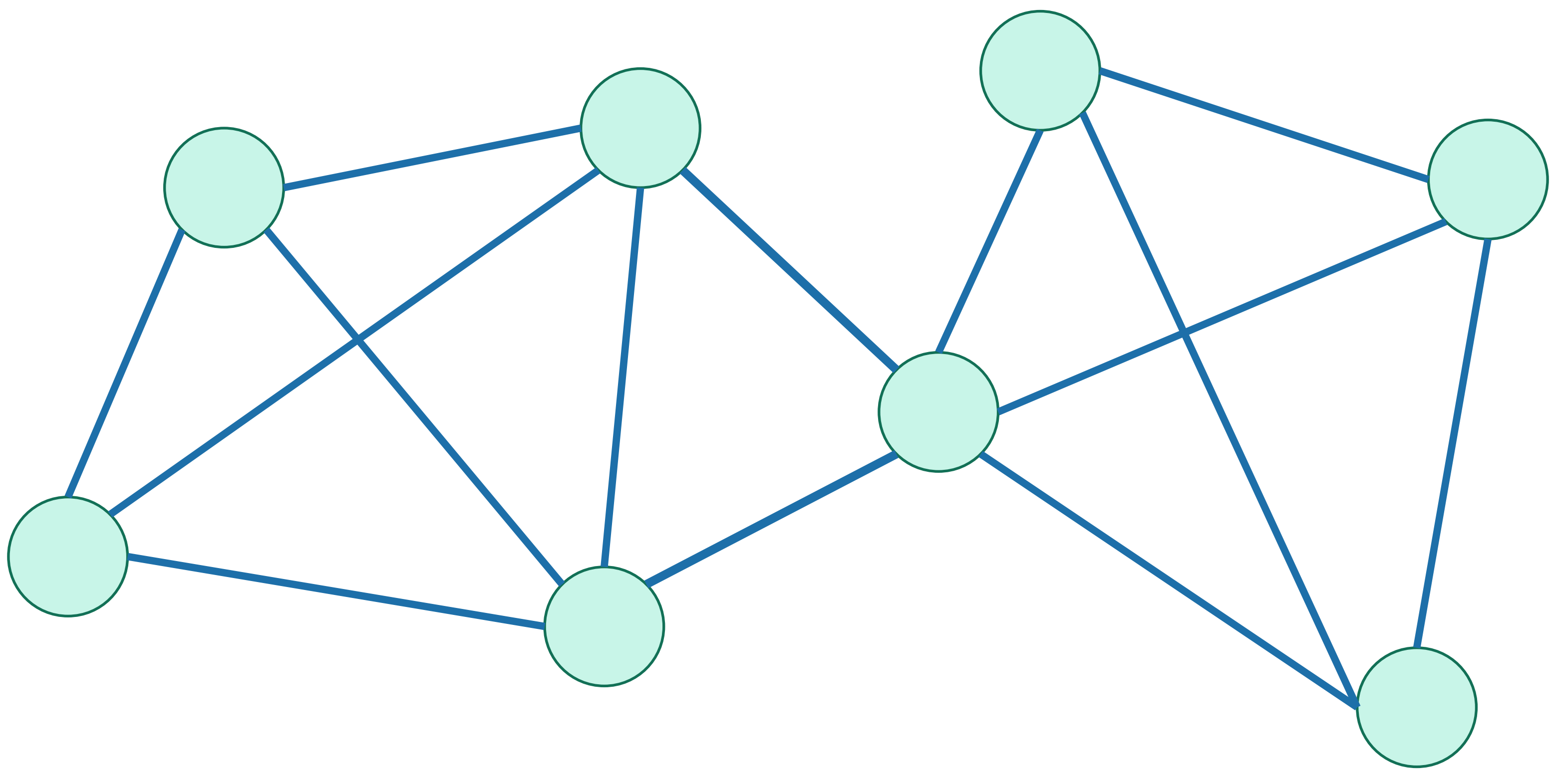
These edges **cross the cut.**

- They go from one part to the other.



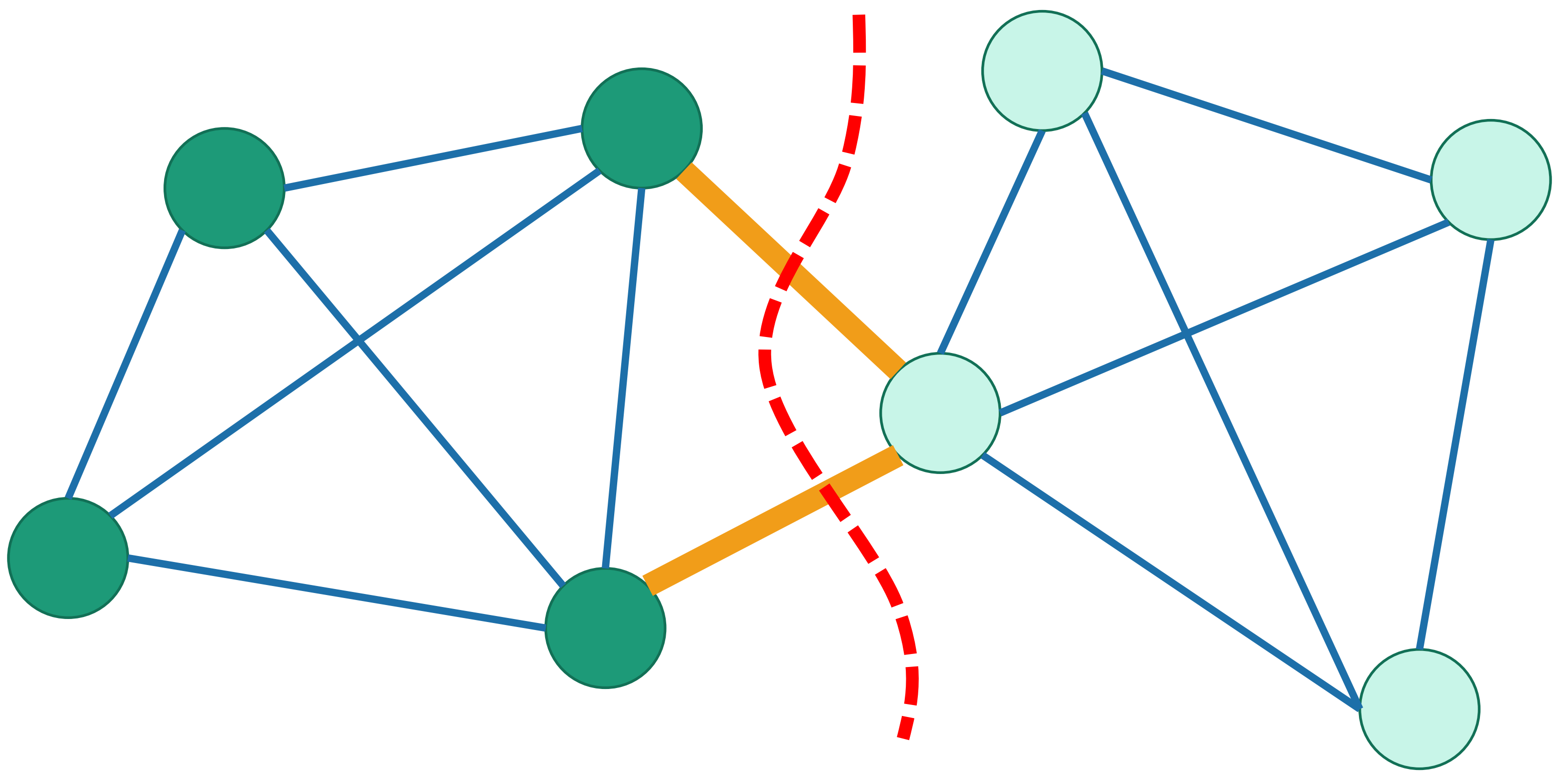
A (global) minimum cut

is a cut that has the fewest edges possible crossing it.



A (global) minimum cut

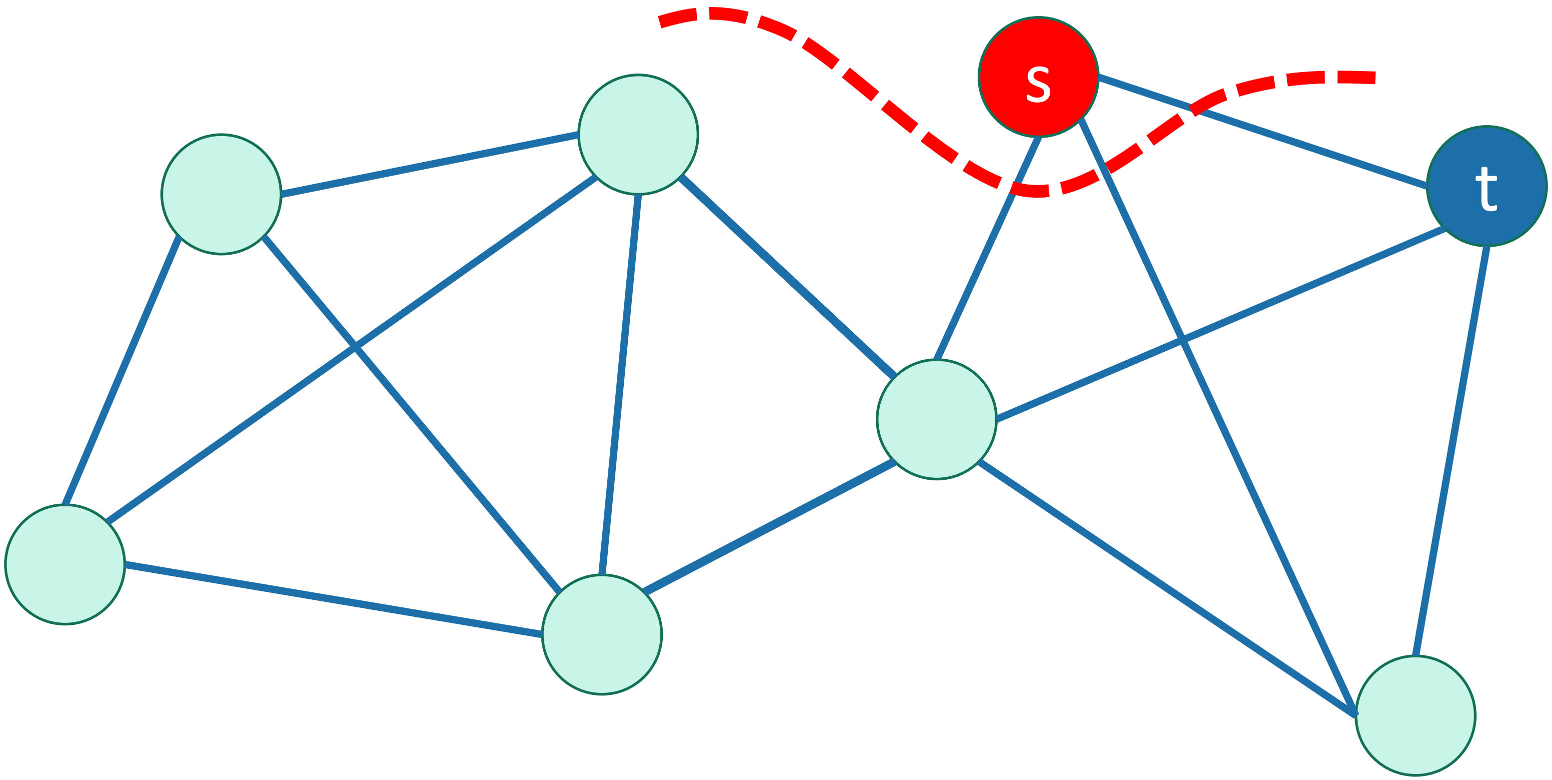
is a cut that has the fewest edges possible crossing it.



Why "global"?

Minimum cut which separates a specified vertex s from t

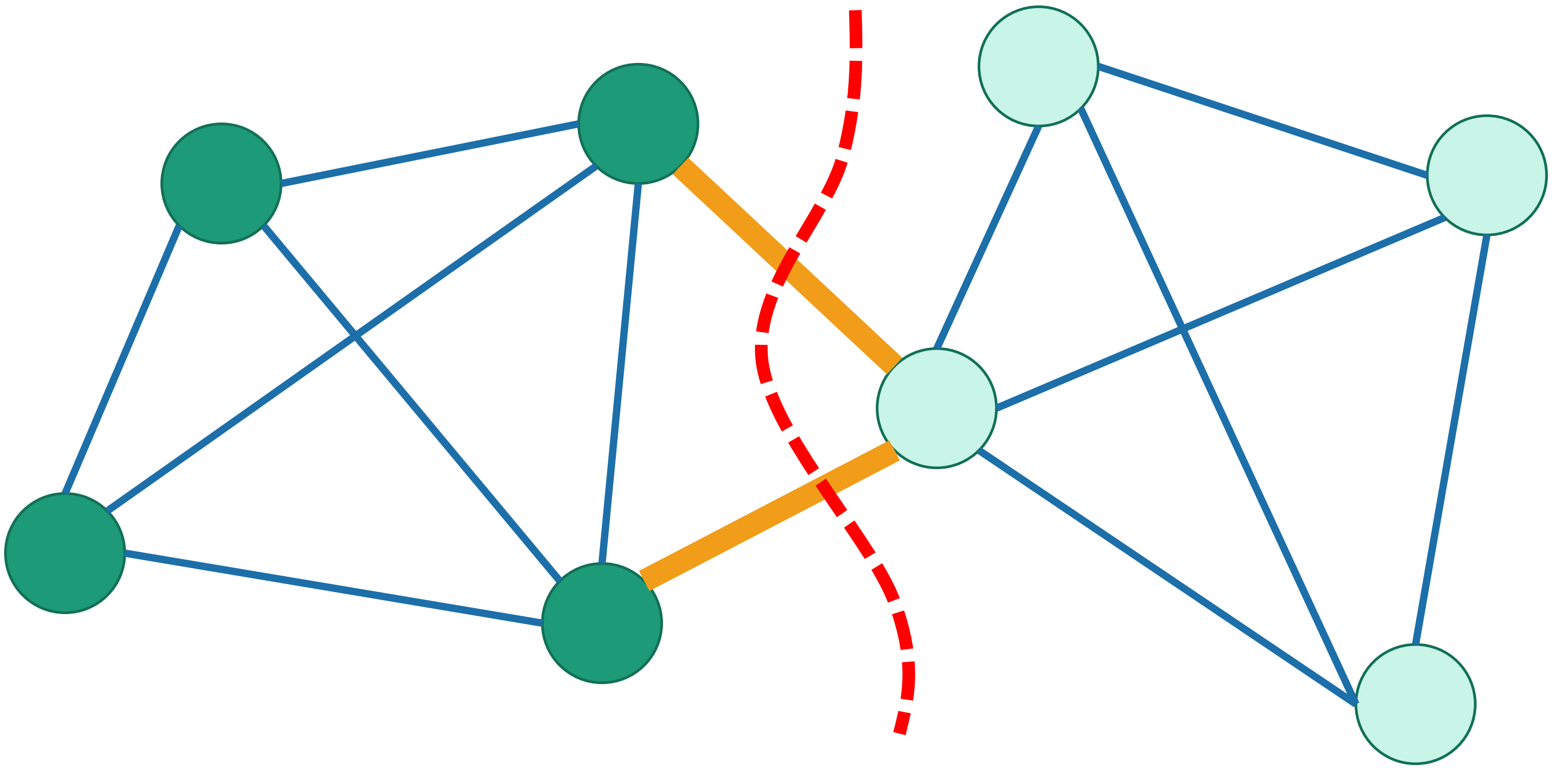
- In the next section, we'll talk about **min s-t cuts**



- At this time, there are no special vertices, so the minimum cut is **"global."**

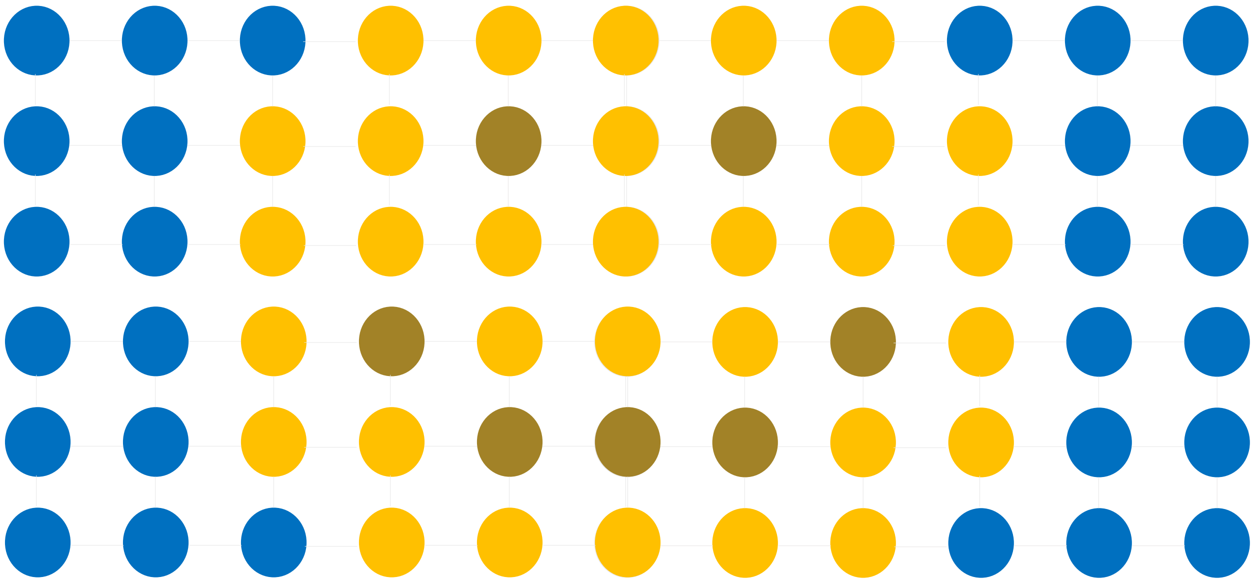
A (global) minimum cut

is a cut that has the fewest edges possible crossing it.



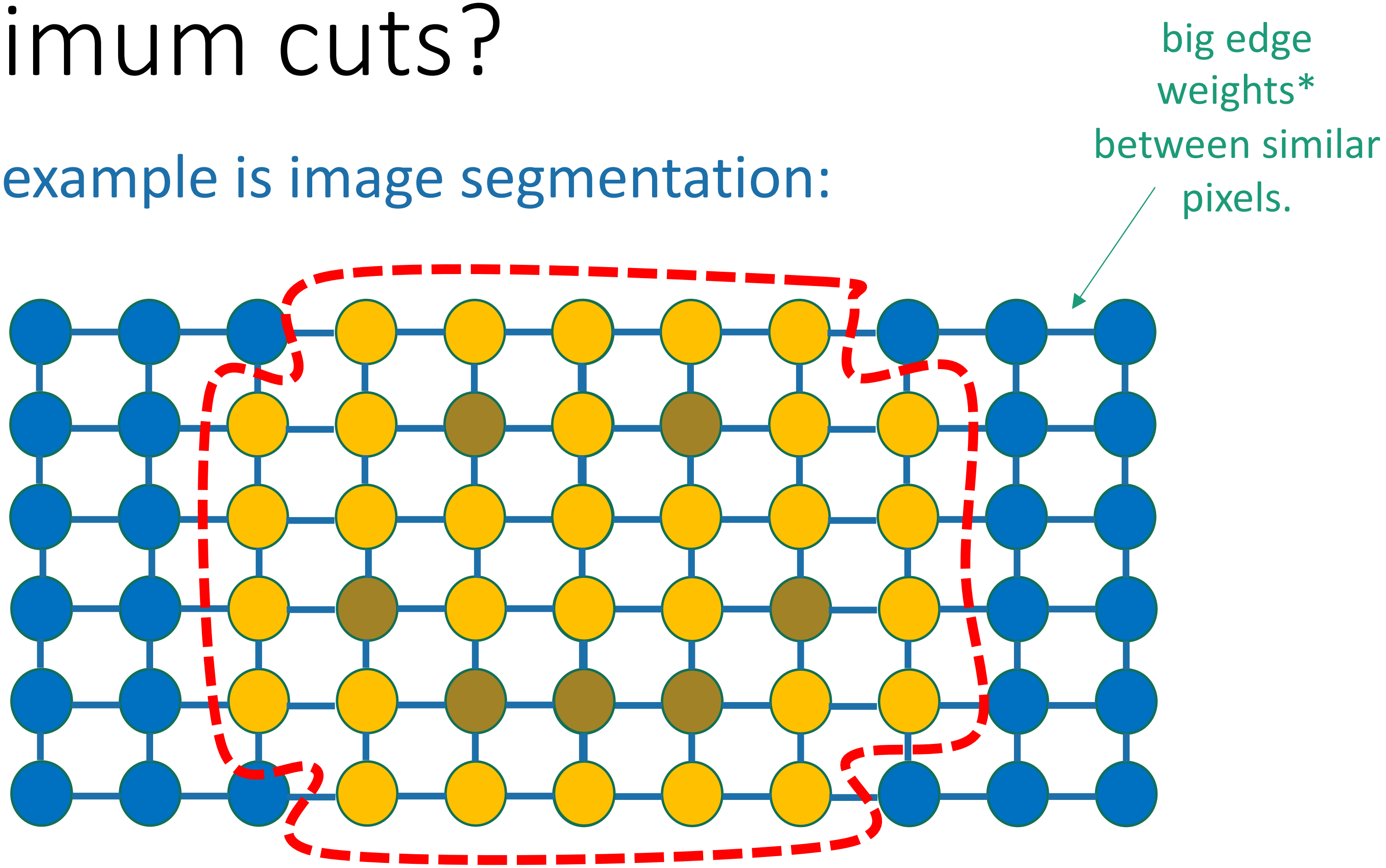
Why might we care about global minimum cuts?

- One example is image segmentation:



Why might we care about global minimum cuts?

- One example is image segmentation:



- We'll see more applications for other sorts of min-cuts soon

Karger’s algorithm

- Finds **global minimum cuts** in undirected graphs
- Randomized algorithm
- Karger’s algorithm **might be wrong**.
  - Compare to QuickSort, which just might be slow.

- Why would we want an algorithm that might be wrong?
  - **With high probability it won’t be wrong.**
  - Maybe the stakes are low and the cost of a deterministic algorithm is high.

Different sorts of gambling

- QuickSort is a **Las Vegas randomized algorithm**
  - It is always correct.
  - It might be slow.

Yes, this is a technical term.

Formally:

- For all inputs A, QuickSort(A) returns a sorted array.
- For all inputs A, with high probability over the choice of pivots, QuickSort(A) runs quickly.

Different sorts of gambling

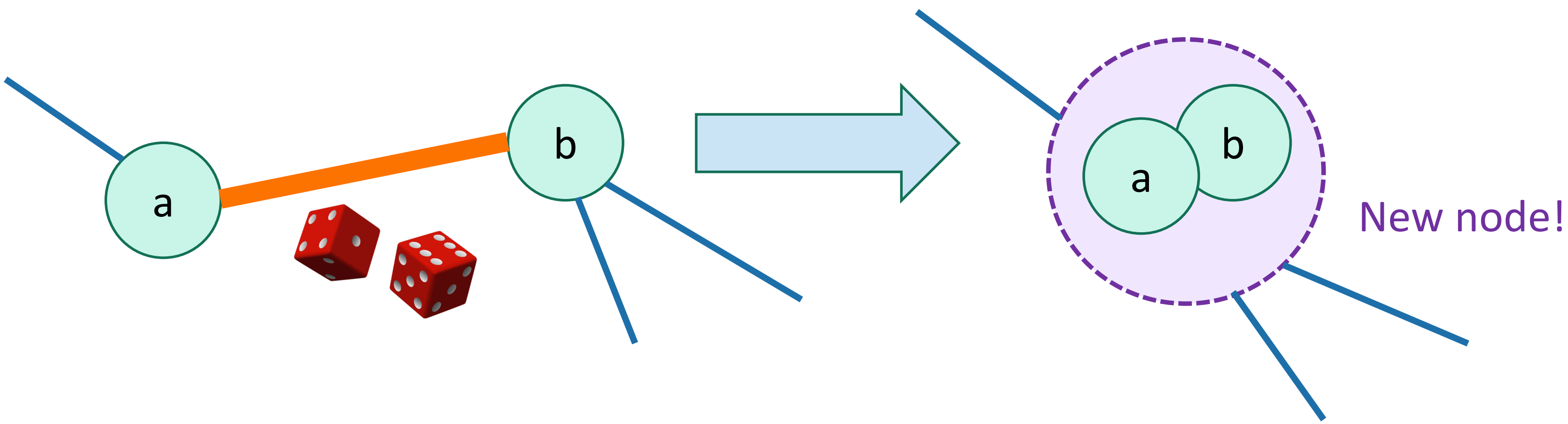
- Karger's Algorithm is a **Monte Carlo randomized algorithm**
  - It is always fast.
  - It might be wrong.

Formally:

- For all inputs G, with probability at least ___ over the randomness in Karger's algorithm, Karger(G) returns a minimum cut.
- For all inputs G, with probability 1 Karger's algorithm runs in time no more than ____.

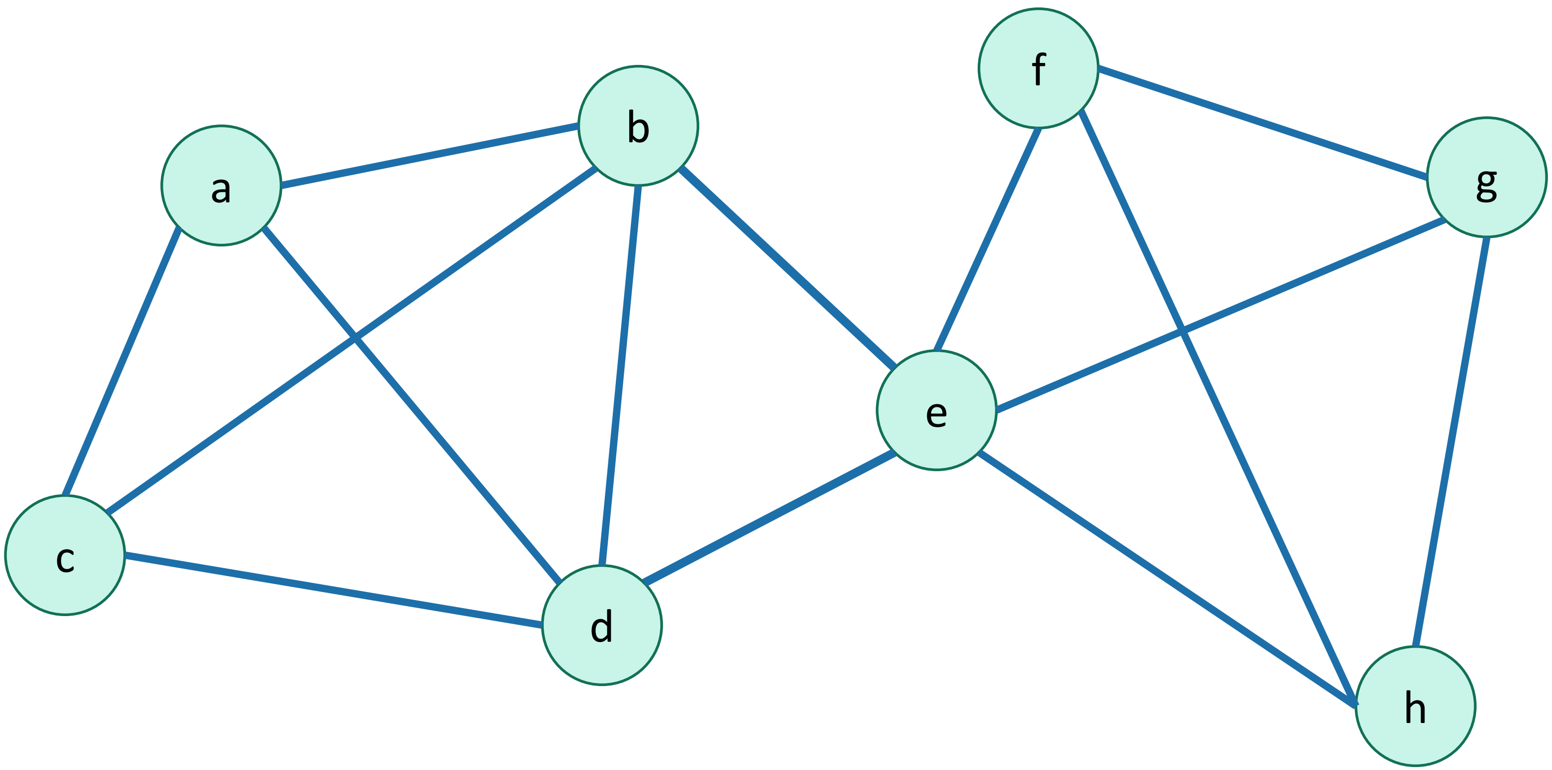
Karger's Algorithm

- Pick a random edge.
- **Contract** it.
- Repeat until you only have two vertices left.

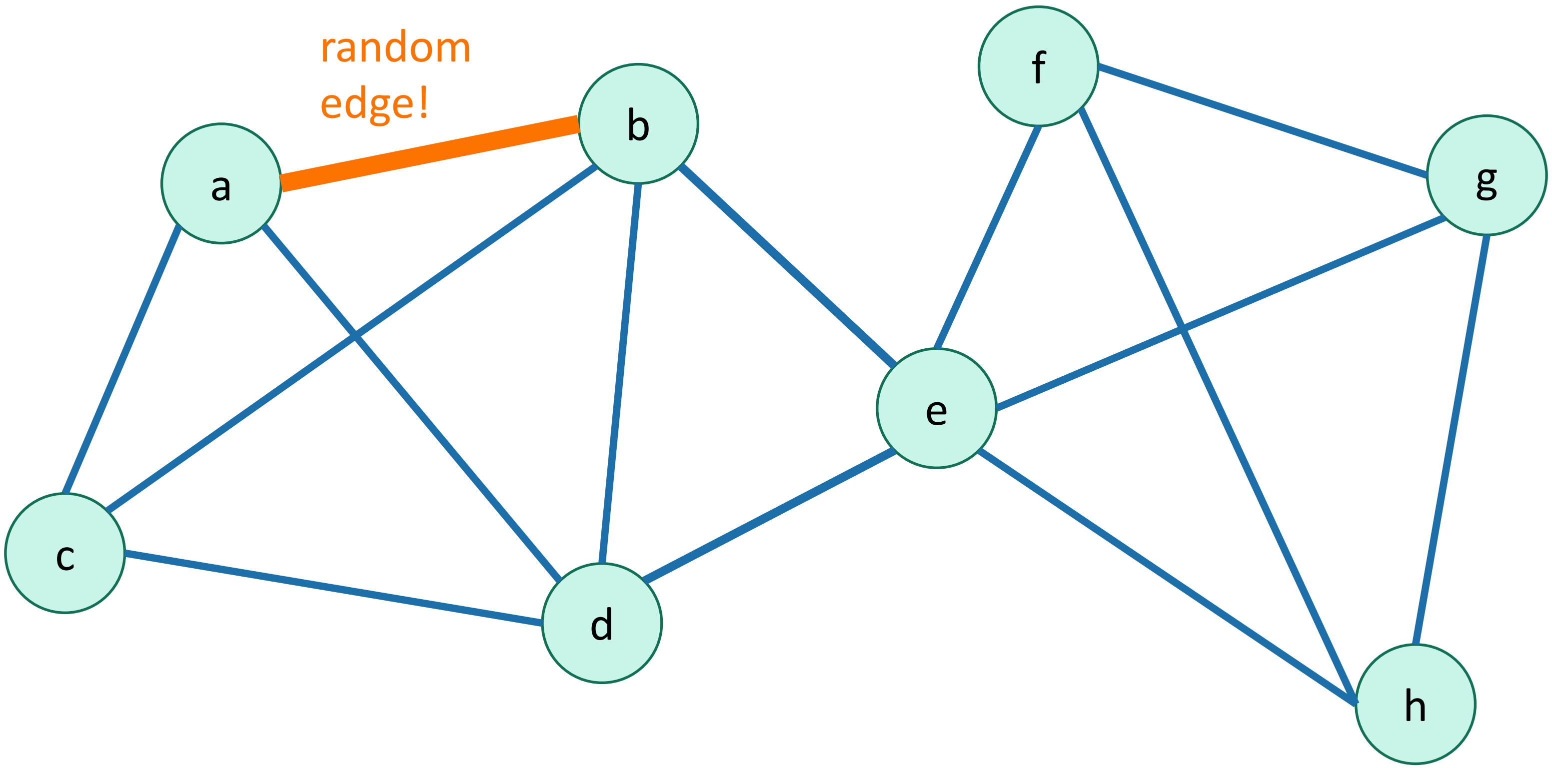


Why is this a good idea? We'll see shortly.

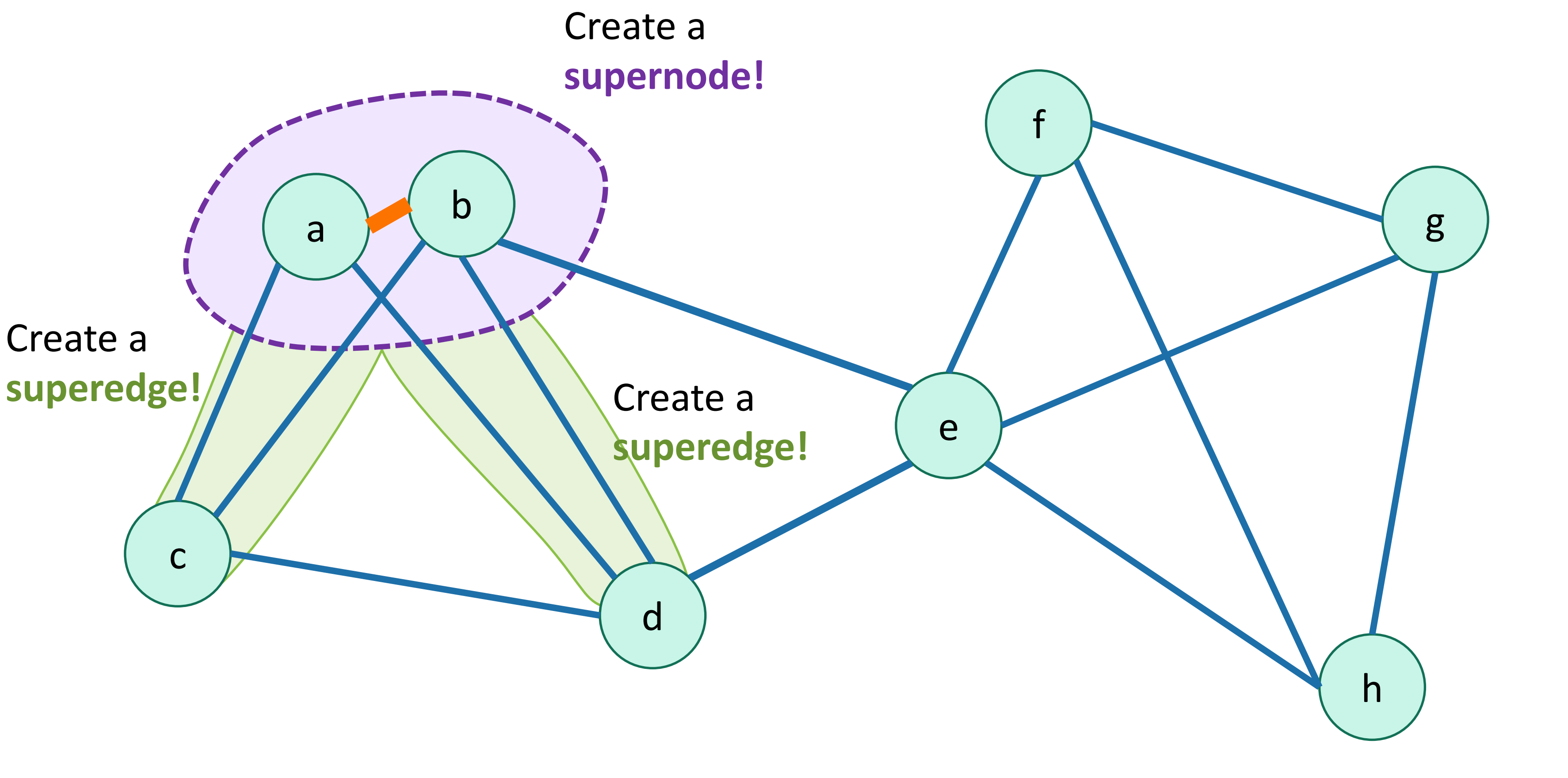
Karger's algorithm



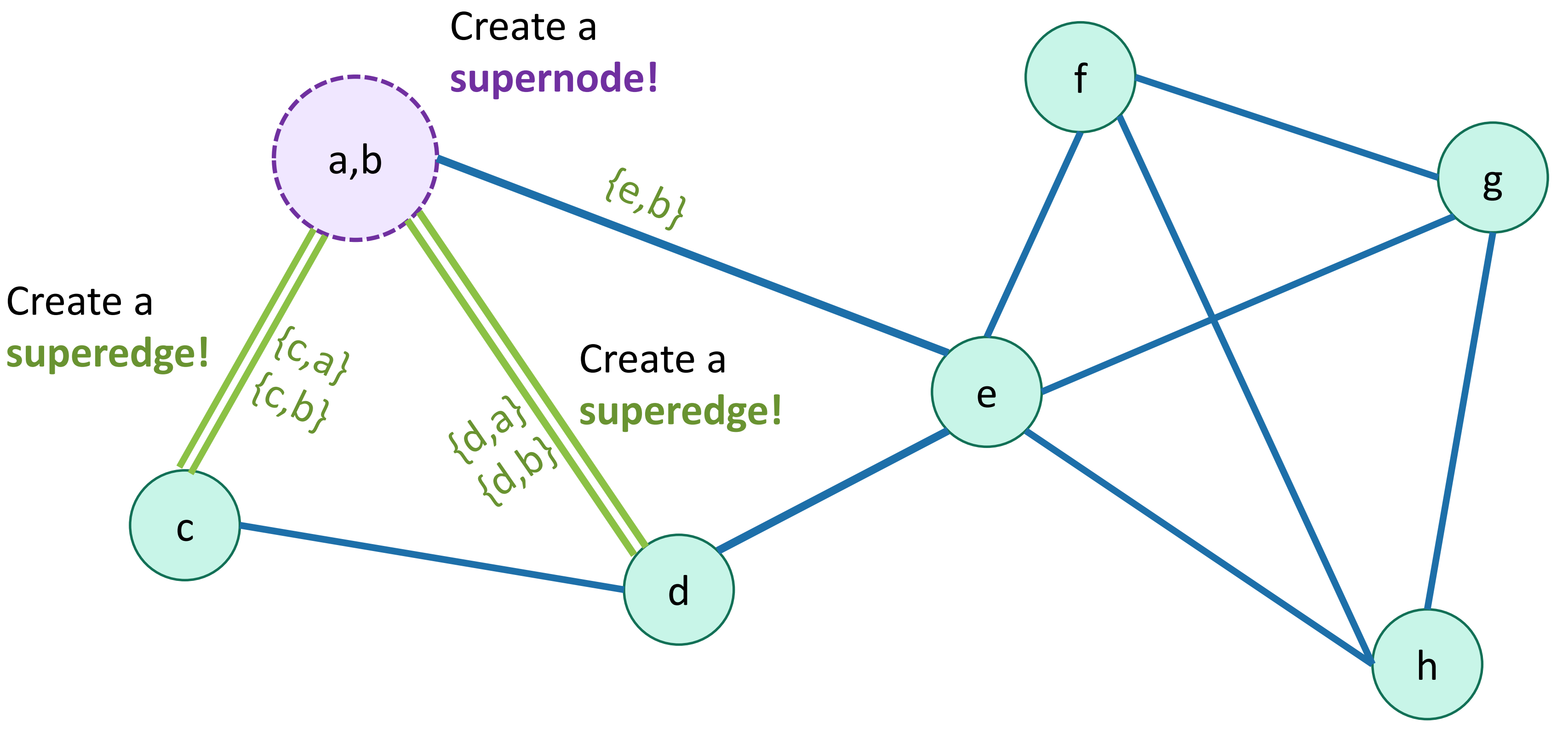
Karger’s algorithm



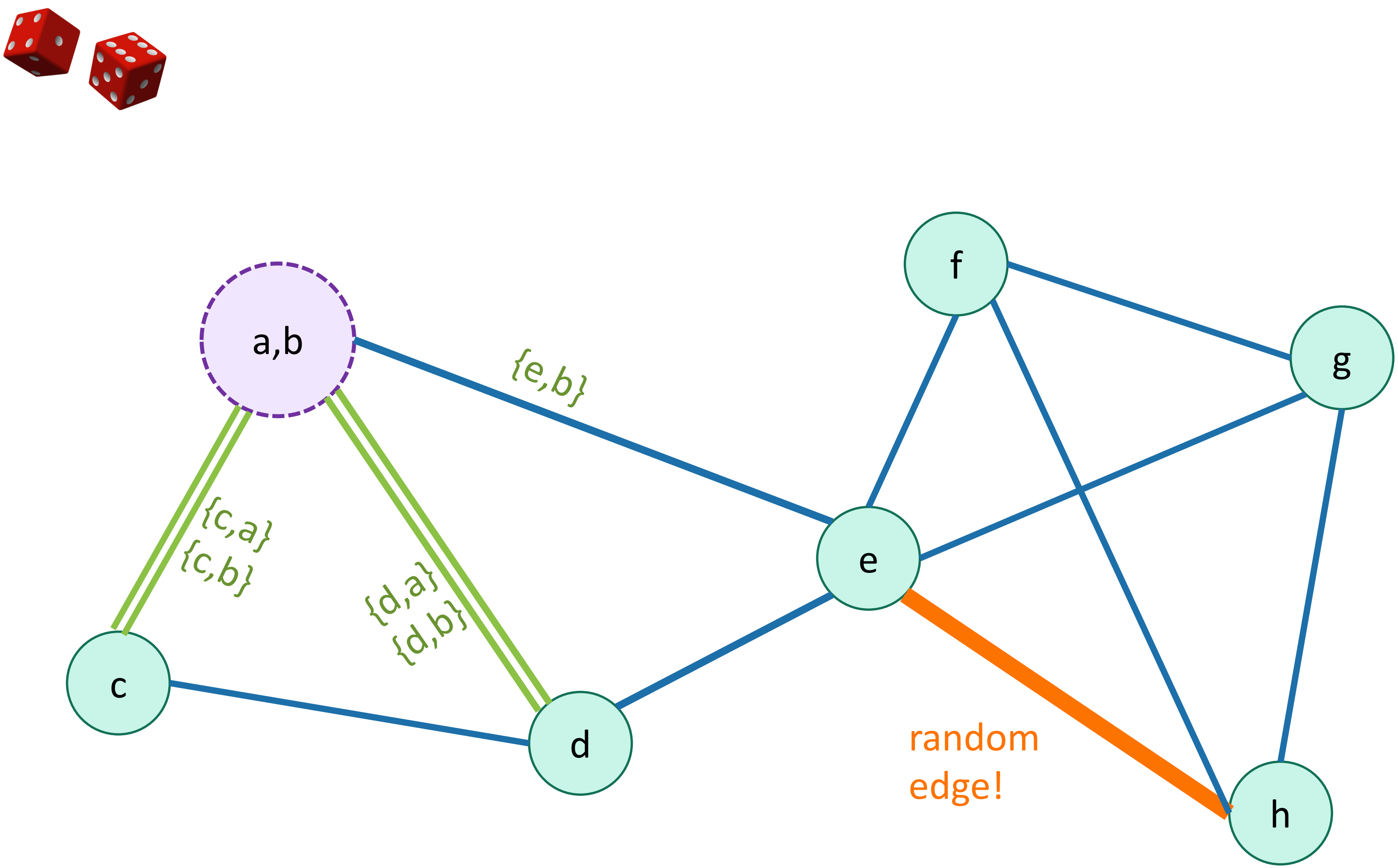
Karger's algorithm



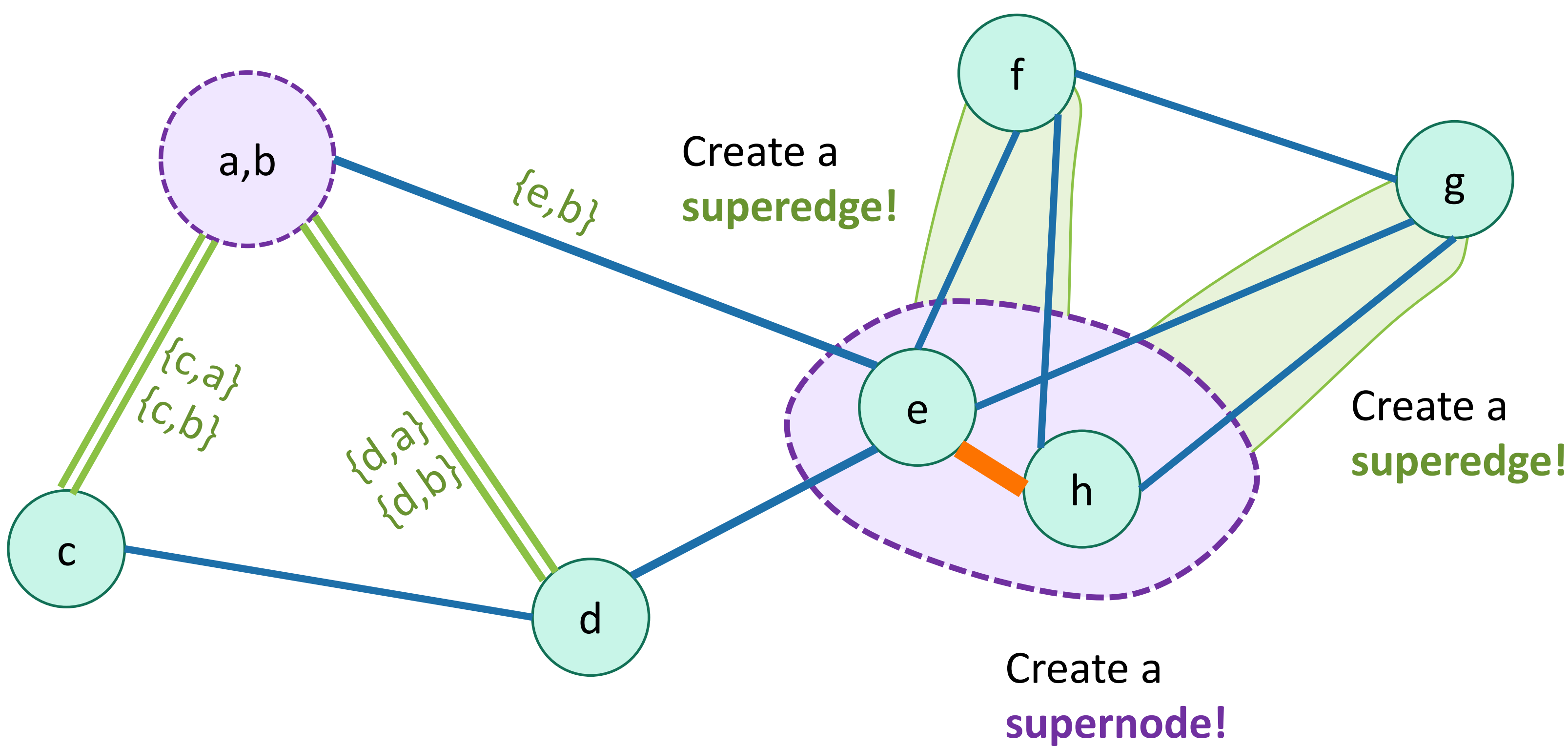
Karger's algorithm



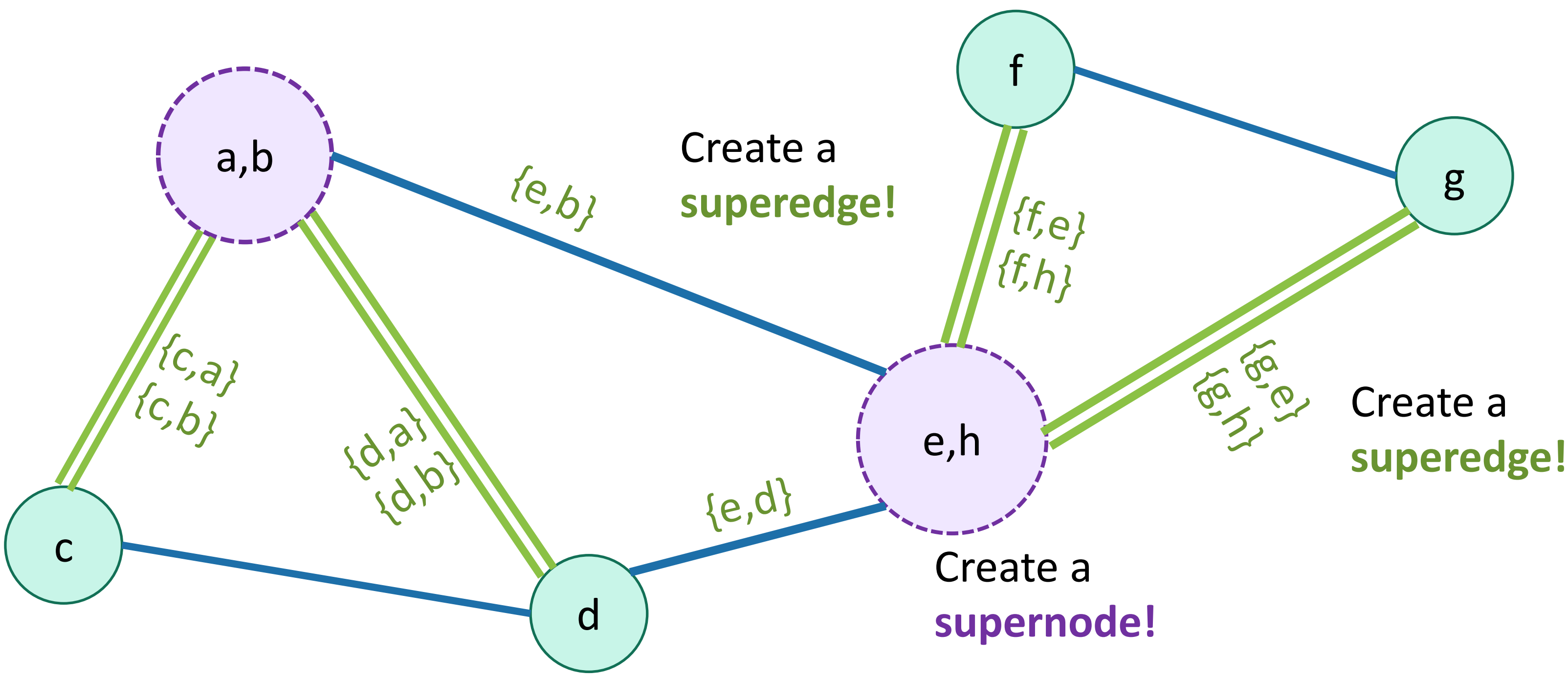
Karger's algorithm



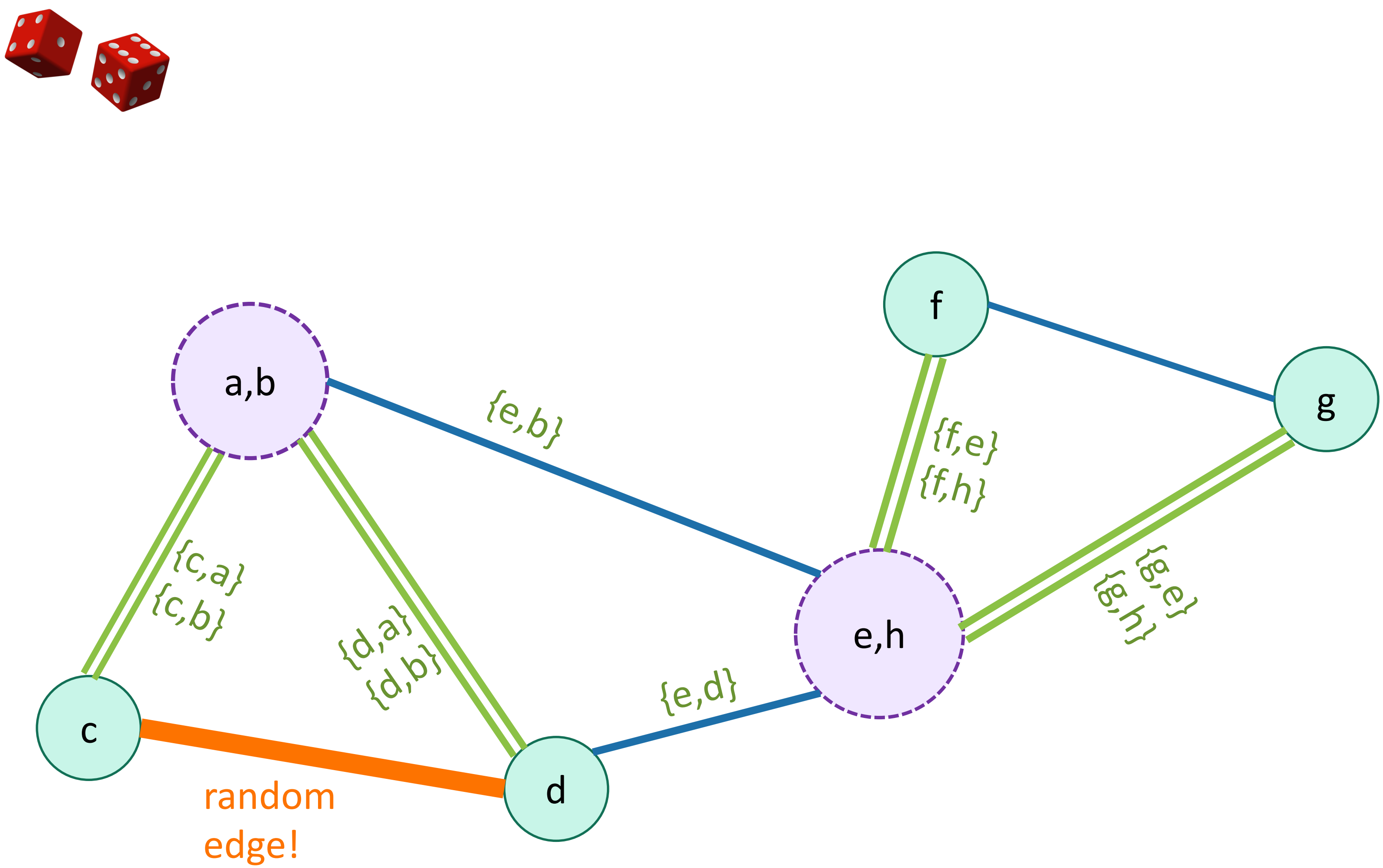
Karger's algorithm



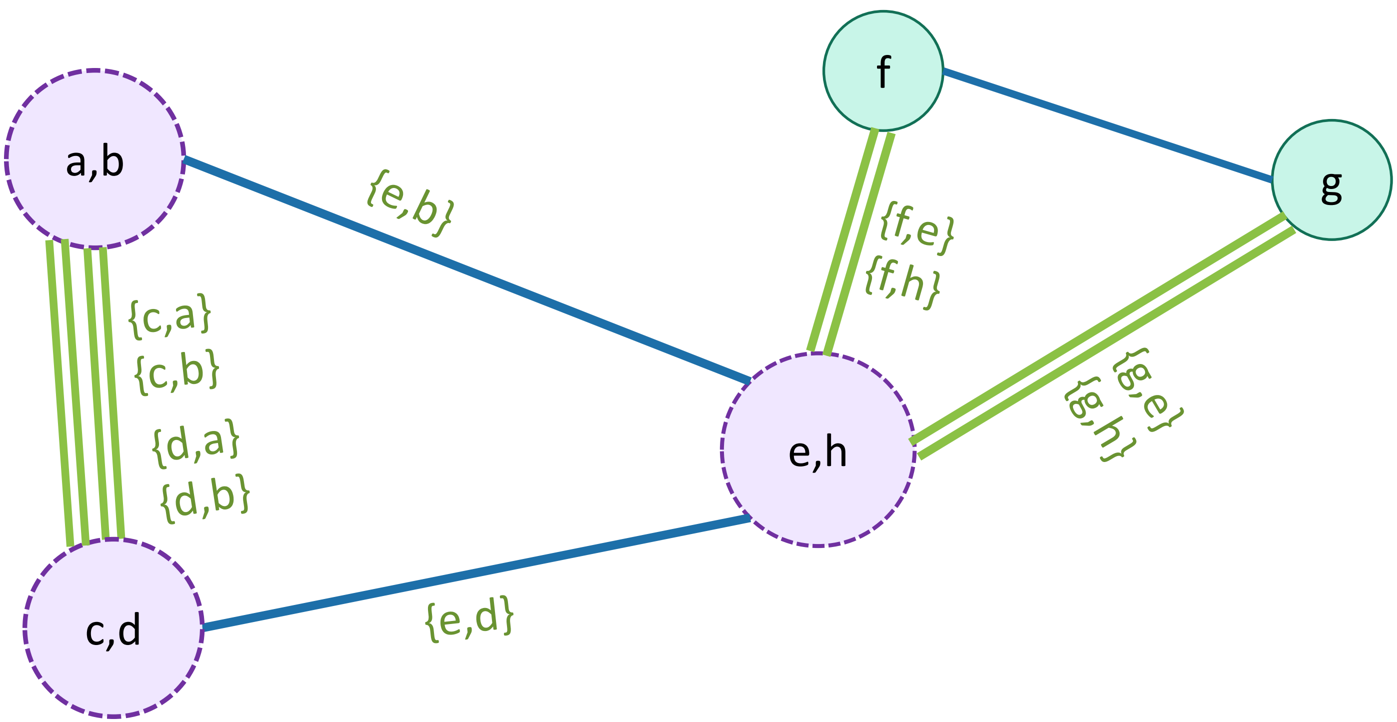
Karger's algorithm



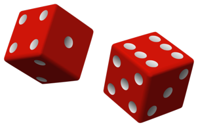
Karger's algorithm

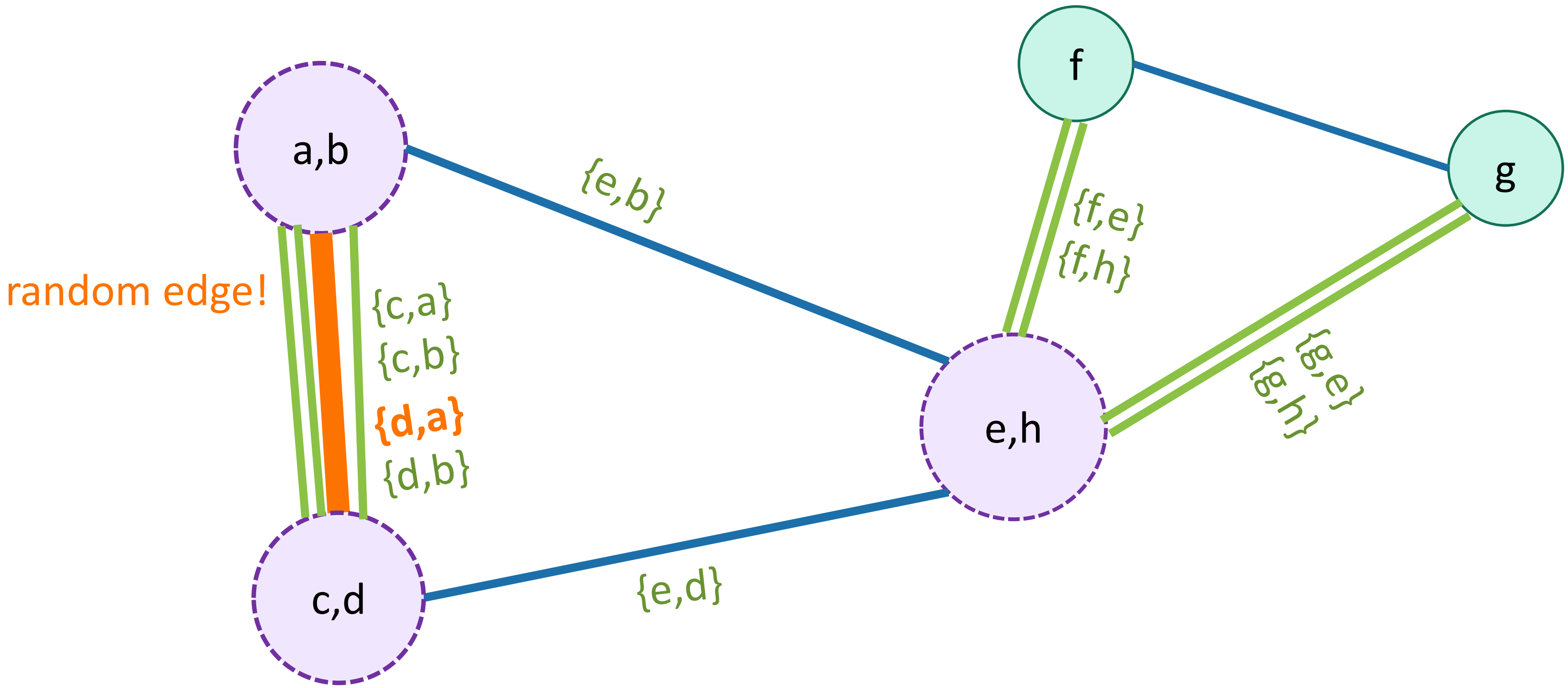


Karger's algorithm

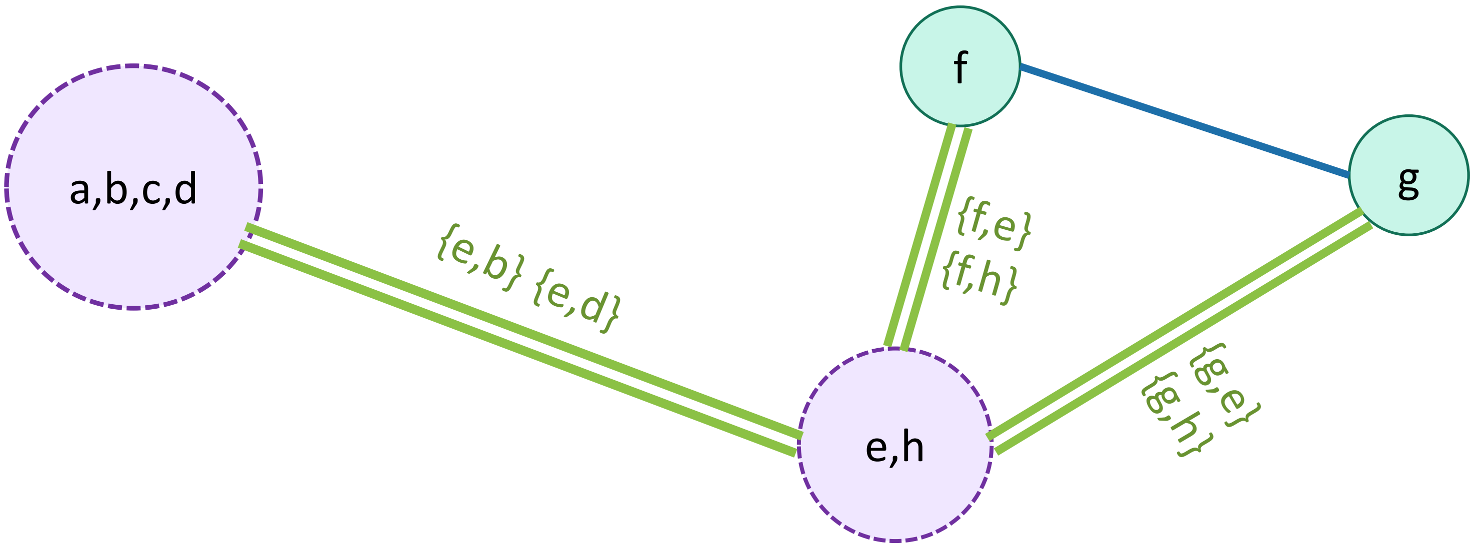


Karger's algorithm

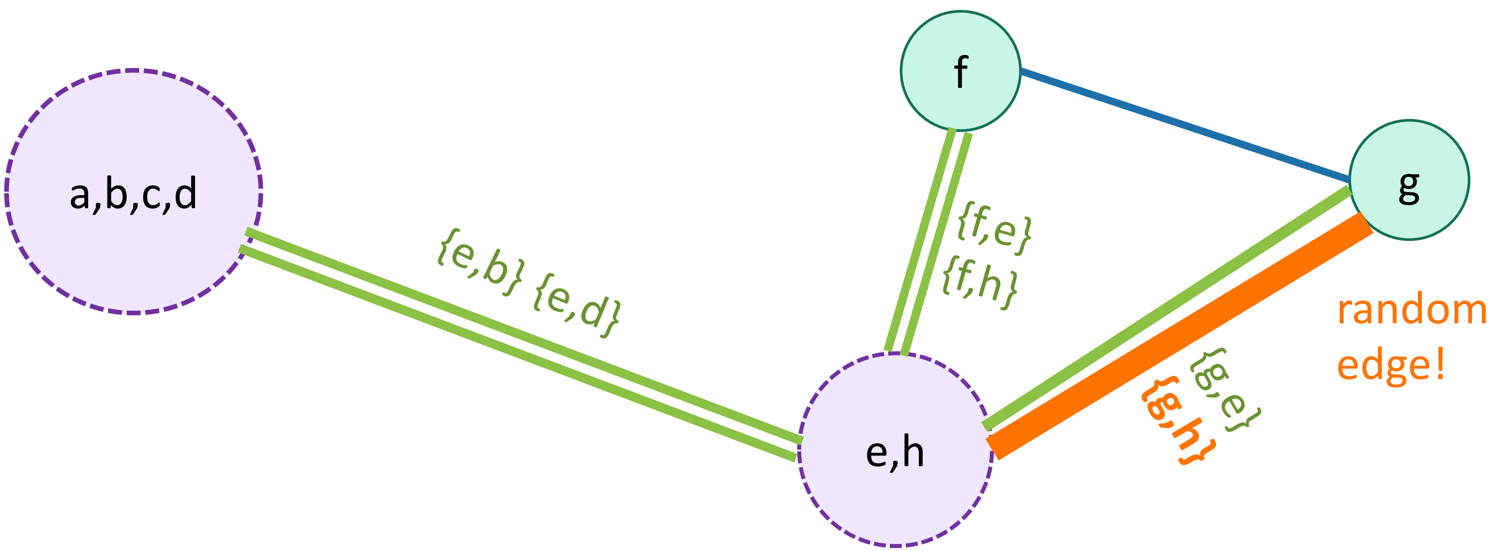




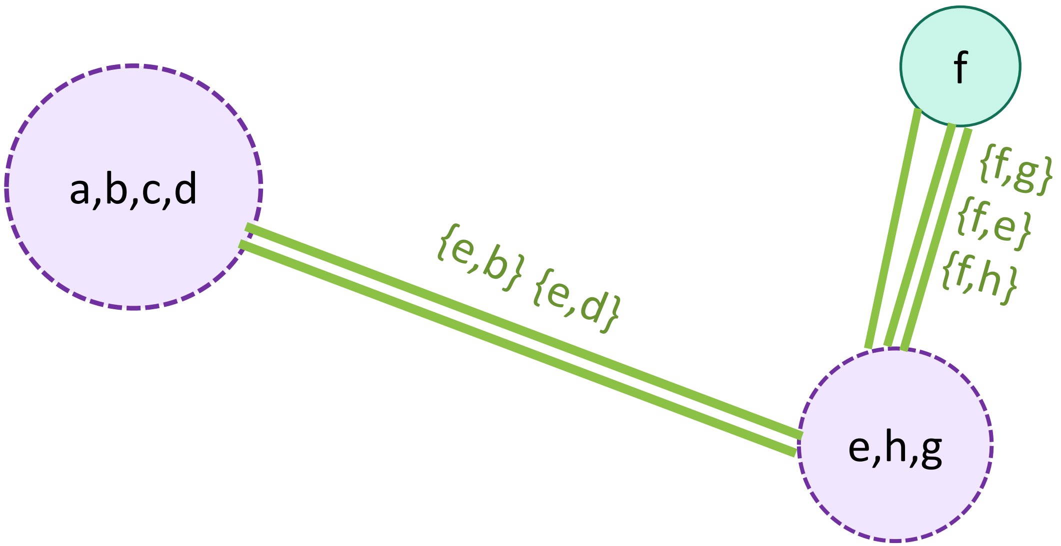
Karger's algorithm



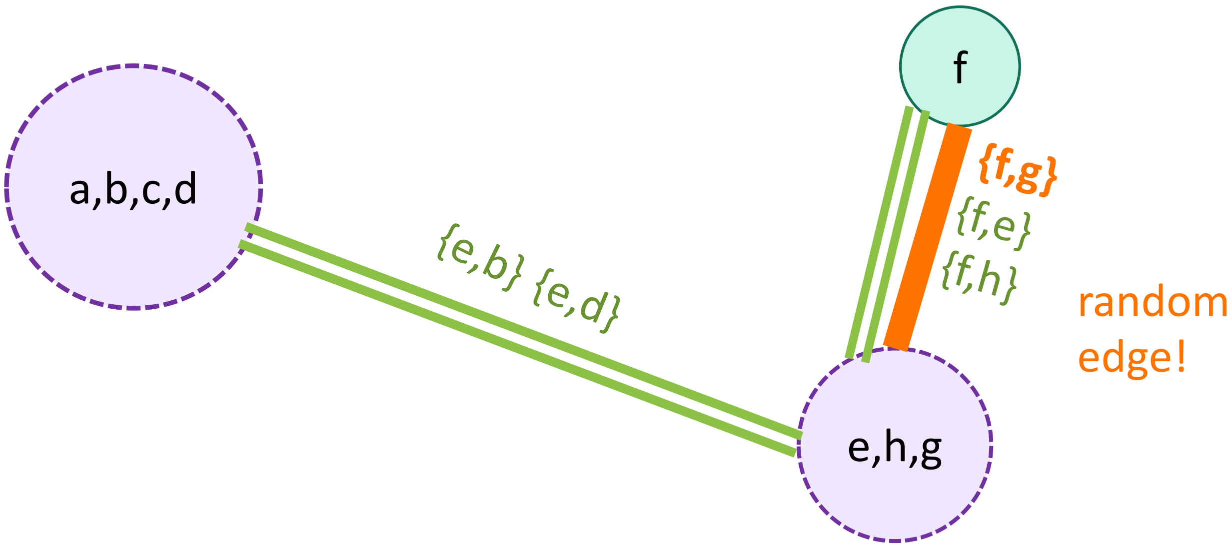
Karger's algorithm



Karger's algorithm



Karger's algorithm



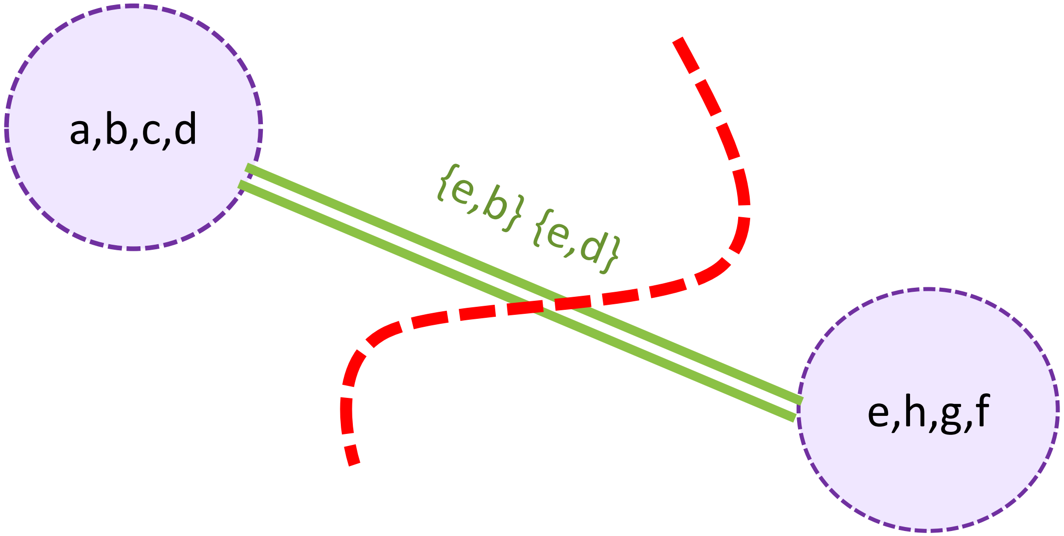
Karger's algorithm

The **minimum cut** is given by the remaining super-nodes:

- **{a,b,c,d} and {e,h,f,g}**

Now stop!

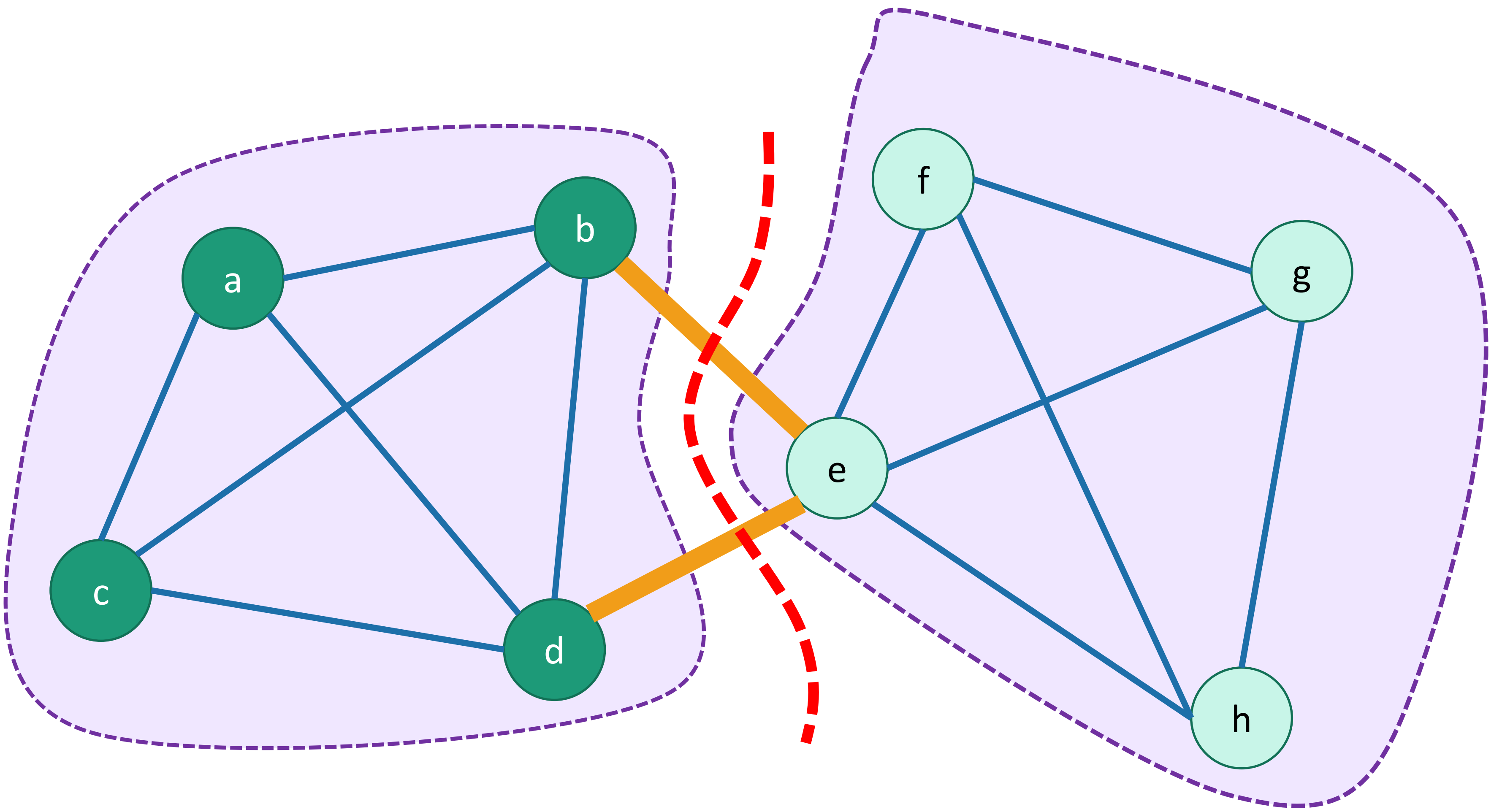
- There are only two nodes left.



Karger’s algorithm

The **minimum cut** is given by the remaining super-nodes:

- **{a,b,c,d} and {e,h,f,g}**



Karger’s algorithm

- Does it work?
- Is it fast?

How do we implement this?

- See Code
  - This maintains a secondary “superGraph” which keeps track of superNodes and superEdges
- Running time?
  - We contract at most n-2 edges
    - Each time we contract an edge we get rid of a vertex, and we get rid of at most n – 2 vertices total.
  - Naively each contraction takes time O(n)
    - Maybe there are about n nodes in the superNodes that we are merging.
  - So total running time O(n^2).
    - We can do $O(m \cdot \alpha(n))$ with a union-find data structure, but $O(n^2)$ is good enough for today.

Pseudocode

Let $\overline{\boldsymbol{u}}$ denote the SuperNode in Γ containing u
Say $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ is the SuperEdge between $\overline{\boldsymbol{u}}, \overline{\boldsymbol{v}}$.

- **Karger**(G=(V,E)):
  - Γ = { SuperNode(v) : v in V } // one supernode for each vertex
  - $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ = {(u,v)} for (u,v) in E // one superedge for each edge
  - $E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$ = {} for (u,v) not in E.
  - F = copy of E // we'll choose randomly from F
  - **while** $|\Gamma| > 2$:
    - (u,v) ← uniformly random edge in F
    - **merge**(u, v)

      // merge the SuperNode containing u with the SuperNode containing v.
    - $F \leftarrow F \setminus E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{v}}}$

      // remove all the edges in the SuperEdge between those SuperNodes.
  - **return** the cut given by the remaining two superNodes.

The **while** loop runs n-2 times

merge takes time O(n) naively

- **merge**(u, v): // merge also knows about Γ and the $E_{\boldsymbol{u},\boldsymbol{v}}$'s
  - $\overline{\boldsymbol{x}}$ = SuperNode($\overline{\boldsymbol{u}} \cup \overline{\boldsymbol{v}}$) // create a new supernode
  - for each **w** in $\Gamma \setminus \{\overline{\boldsymbol{u}}, \overline{\boldsymbol{v}}\}$:
    - $E_{\overline{\boldsymbol{x}},\overline{\boldsymbol{w}}} = E_{\overline{\boldsymbol{u}},\overline{\boldsymbol{w}}} \cup E_{\overline{\boldsymbol{v}},\overline{\boldsymbol{w}}}$
  - Remove $\overline{\boldsymbol{u}}$ and $\overline{\boldsymbol{v}}$ from Γ and add $\overline{\boldsymbol{x}}$.

total runtime O(n^2)

We can do a bit better with fancy data structures, but let's go with this for now.

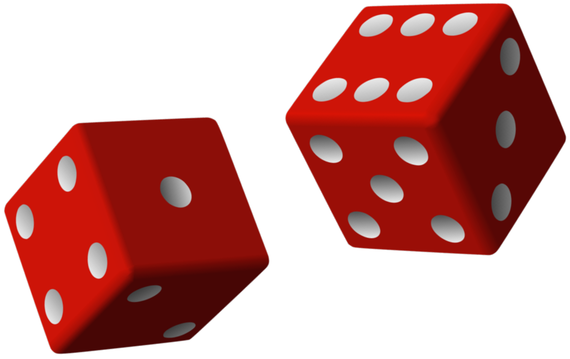
Karger’s algorithm

- Does it work?
  - No?

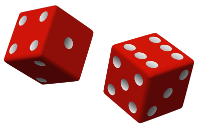
- Is it fast?
  - $O(n^2)$

Why did that work?

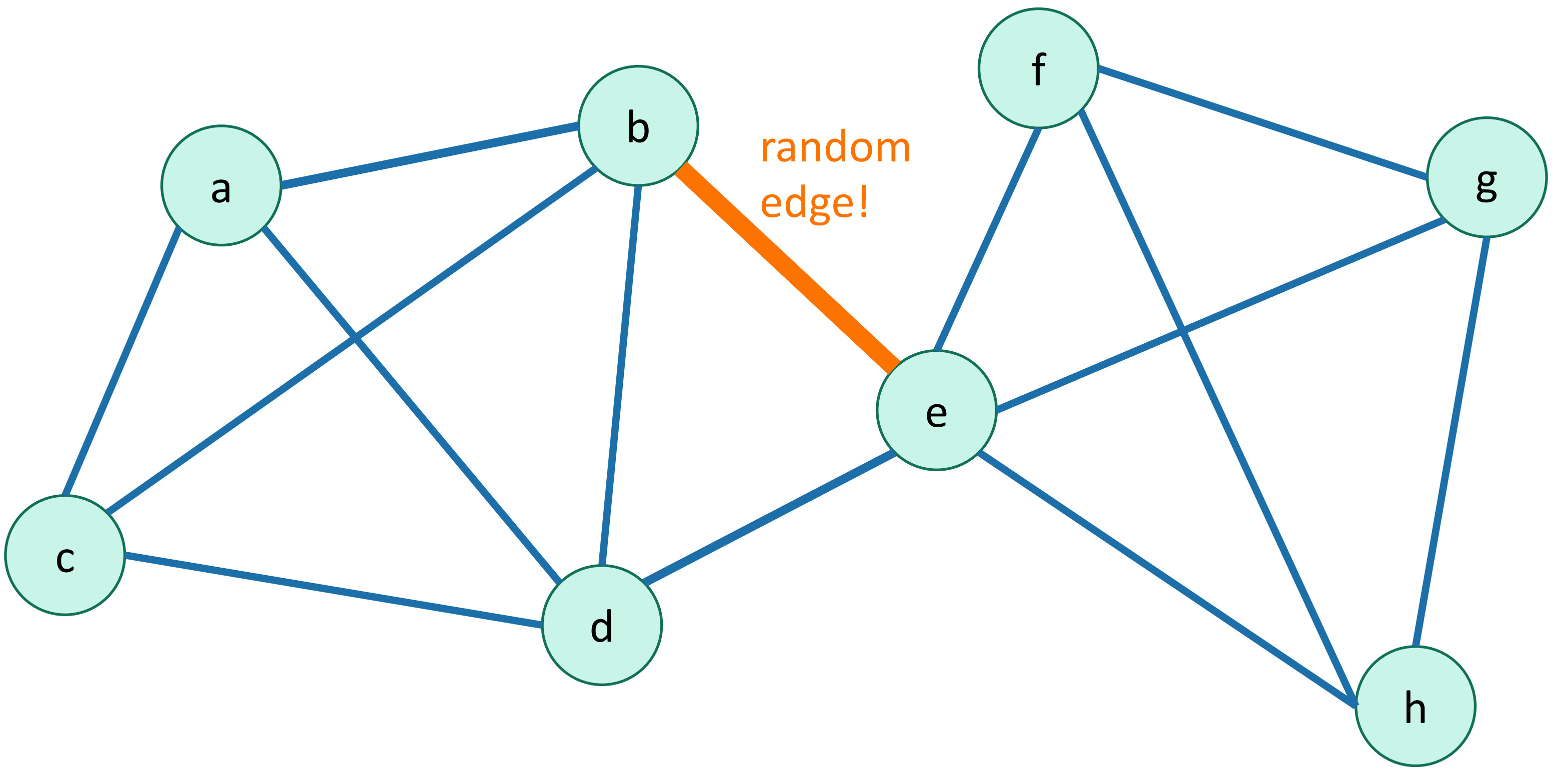
- We got really lucky!
- This could have gone wrong in so many ways.



Karger's algorithm



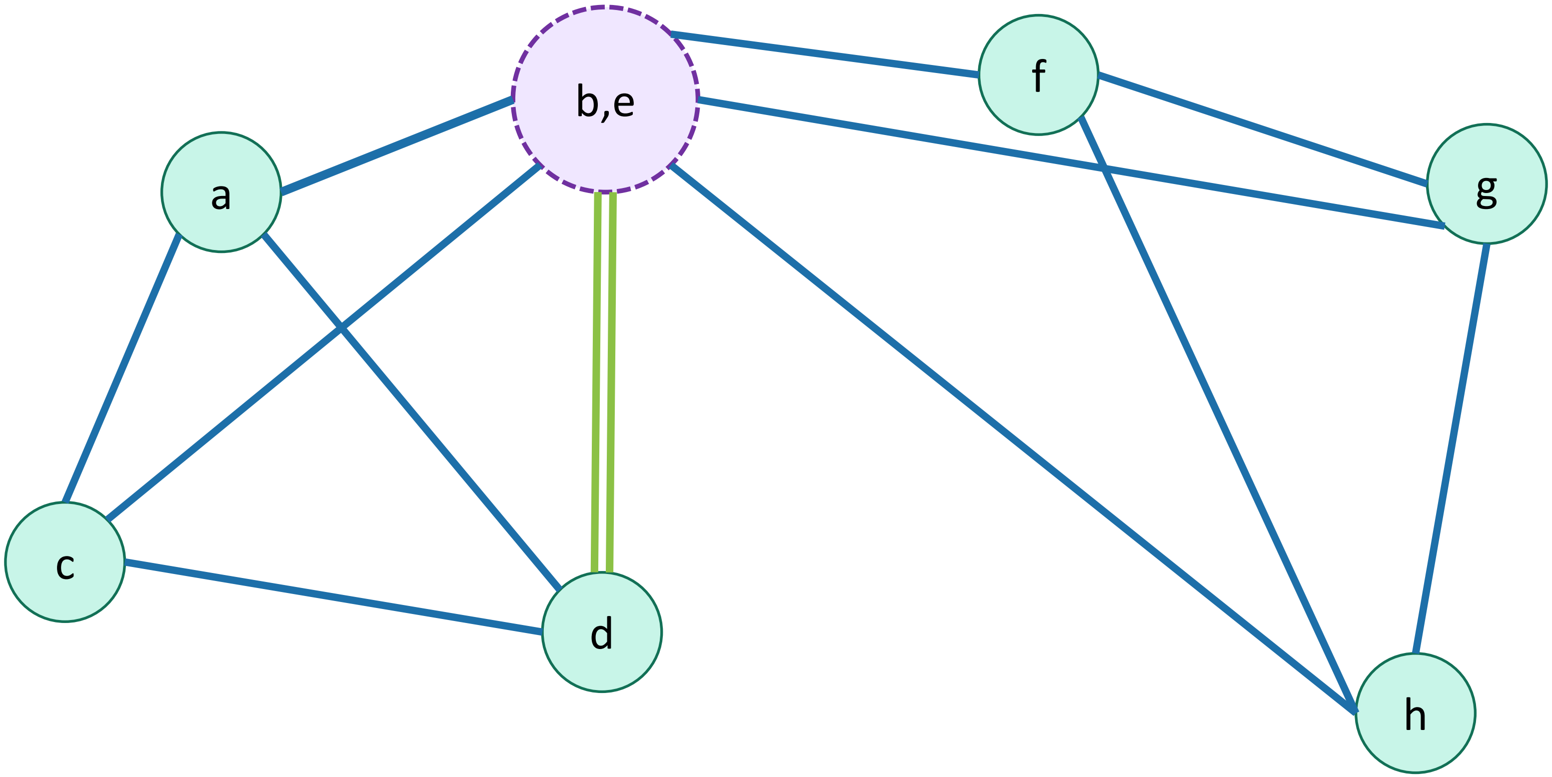
Say we had chosen this edge



Karger's algorithm

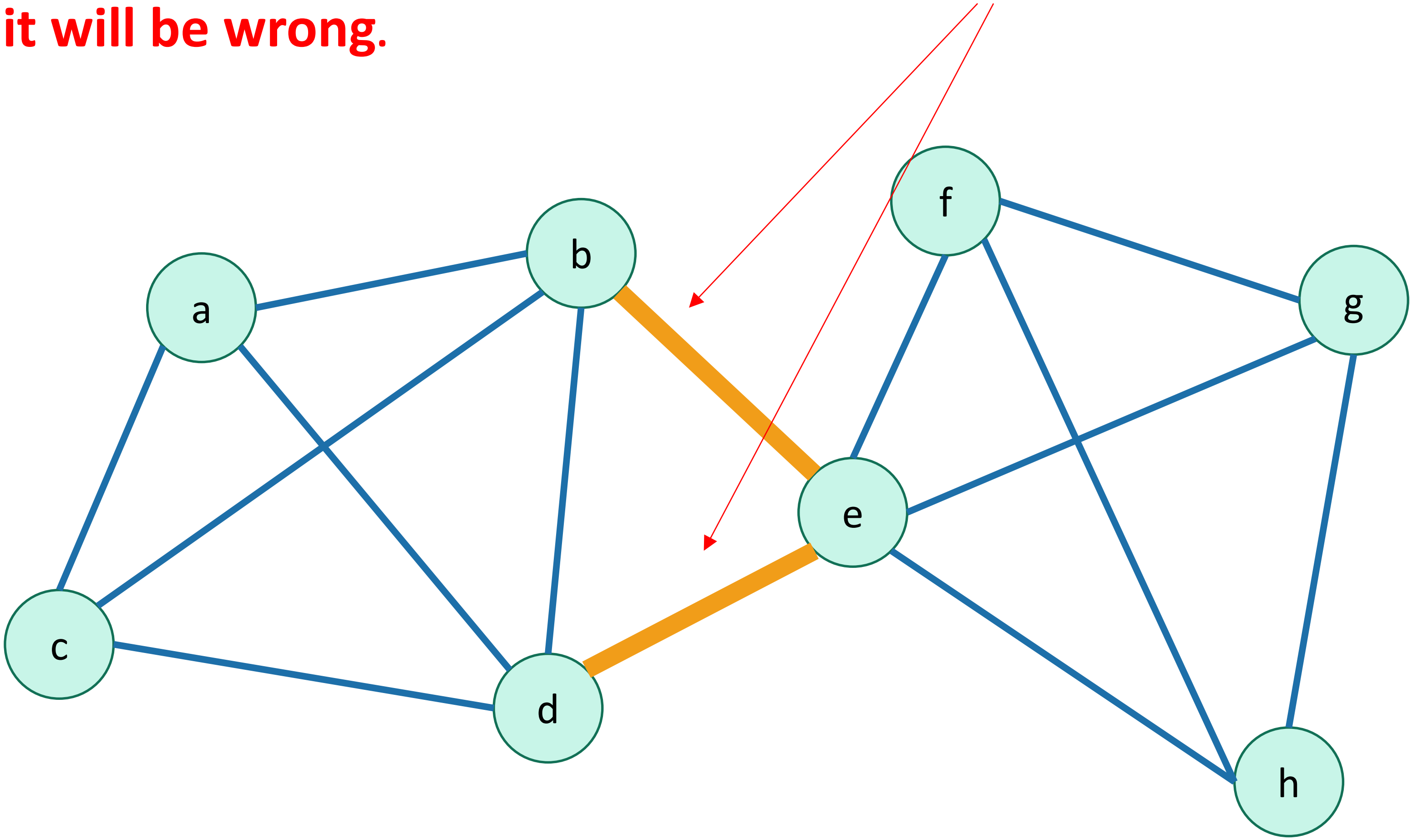
Say we had chosen this edge

Now there is **no way** we could return a cut that separates b and e.

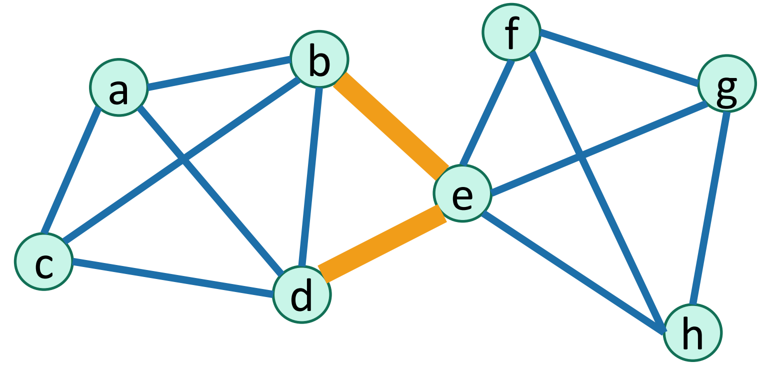


Even worse

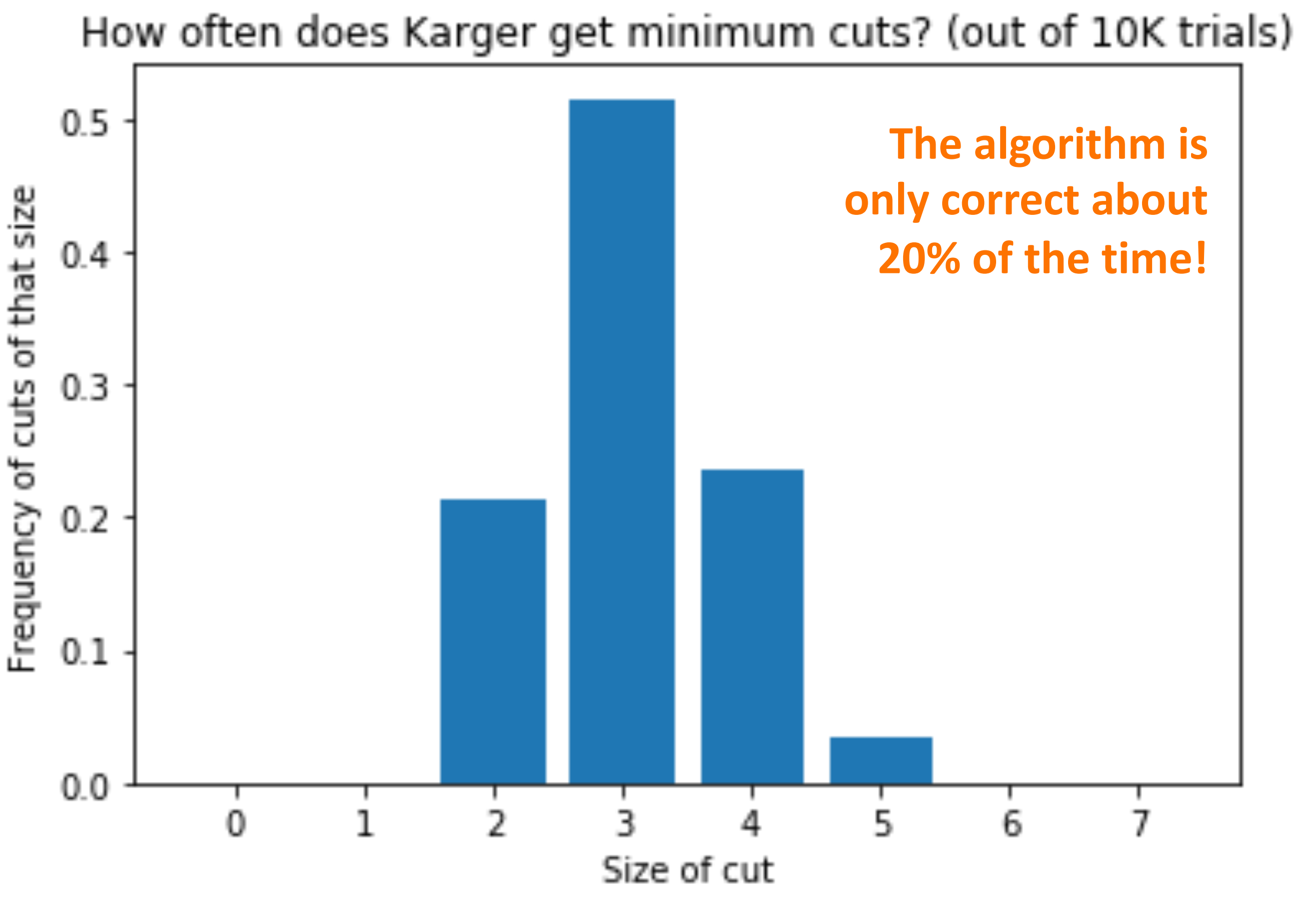
If the algorithm **EVER** chooses either of **these edges**, **it will be wrong**.



How likely is that?

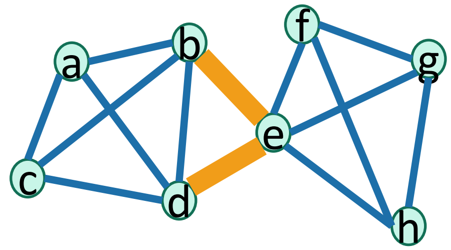


- For this particular graph, we did it 10,000 times:



That doesn't sound good

- Too see why it's good after all, we'll do a case study of this graph.



- Let's compare Karger's algorithm to the algorithm:

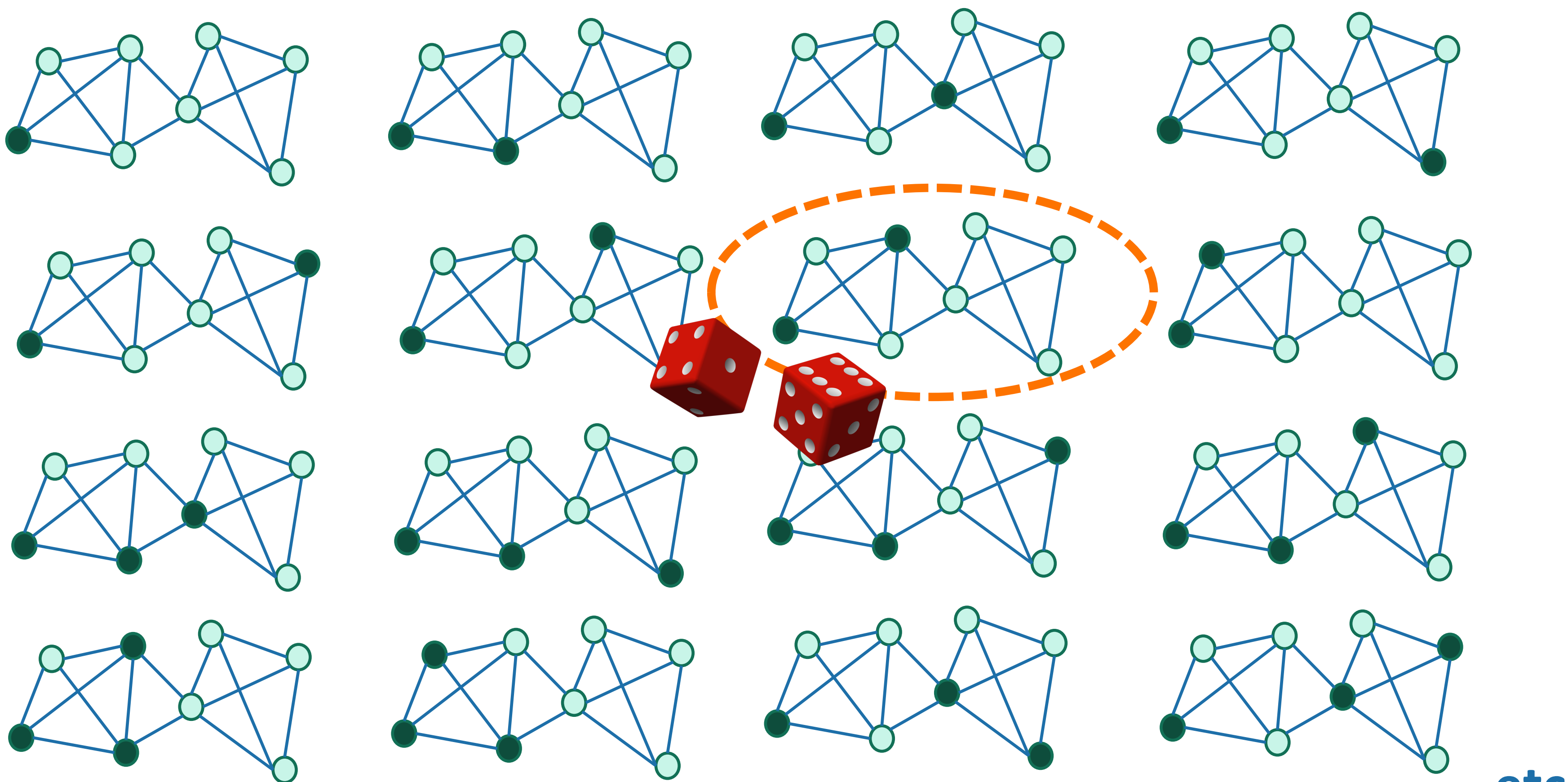
Choose a completely random cut and hope that it's a minimum cut.

The plan:

- See that 20% chance of correctness is actually nontrivial.
- Use repetition to boost an algorithm that's correct 20% of the time to an algorithm that's correct 99% of the time.

Random cuts

- Suppose that we chose cuts uniformly at random.
  - That is, pick a random way to split the vertices into 2 parts.



Random cuts

- Suppose that we chose cuts uniformly at random.
  - That is, pick a random way to split the vertices into 2 parts.

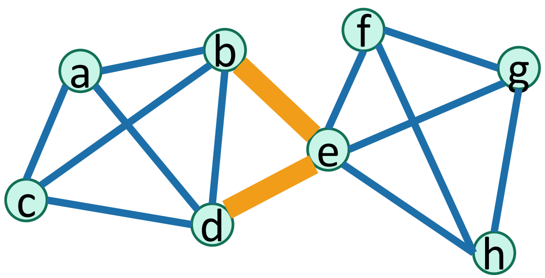
- The probability of choosing the minimum cut is*…

$$\frac{\text{number of min cuts in that graph}}{\text{number of ways to split 8 vertices in 2 parts}} = \frac{2}{2^8 - 2} \approx 0.008$$

- Aka, we get a minimum cut 0.8% of the time.

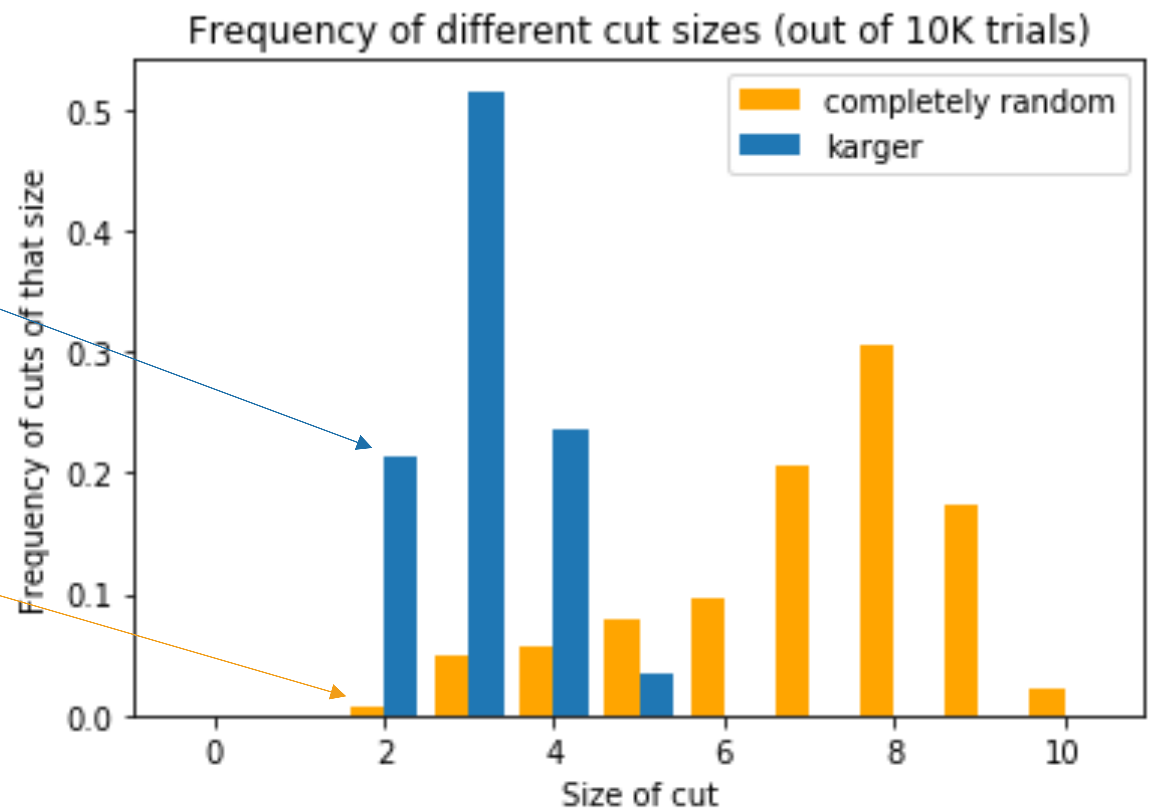
*For this example in particular

Karger is better than completely random!



Karger's alg. is correct about 20% of the time

Completely random is correct about 0.8% of the time



Why does that help?

- Okay, so it's better than random…
- We're still wrong about 80% of the time.
- The main idea: **repeat!**
  - If I'm wrong 20% of the time, then if I repeat it a few times I'll eventually get it right.

The plan:

- See that 20% chance of correctness is actually nontrivial.
- Use repetition to boost an algorithm that's correct 20% of the time to an algorithm that's correct 99% of the time.

Thought experiment

- Suppose you have a magic button that produces one of 5 numbers, {a,b,c,d,e}, uniformly at random when you push it.
- Q: What is the minimum of a,b,c,d,e?

6 3 3 2 5 2 5

How many times do you have to push the button before you see the minimum value?

What is the probability that you have to push it more than 5 times? 10 times?

Binomial Distribution

- E[Number of times we push the button until we get the minimum value] = 1/(0.20) = 5

- Pr[We push the button t times and don't ever get the min] = $(1 - 0.2)^t$

- Pr[We push the button 5 times and don't ever get the min] = $(1 - 0.2)^5 \approx 0.33$

- Pr[We push the button 10 times and don't ever get the min] = $(1 - 0.2)^{10} \approx 0.1$

In this context

- Run Karger's! The cut size is 6!

- Run Karger's! The cut size is 3!

- Run Karger's! The cut size is 3!

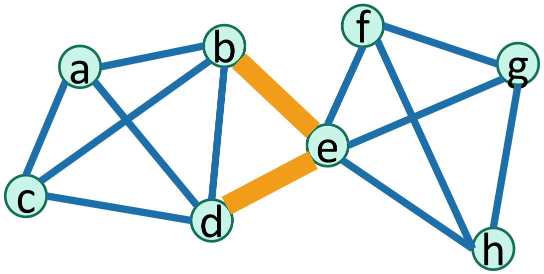
- Run Karger's! The cut size is 2!

Correct!

- Run Karger's! The cut size is 5!

If the success probability is about 20%, then if you run Karger's algorithm 5 times and take the best answer you get, that will likely be correct!

For this particular graph



- Repeat Karger's algorithm about 5 times, and we will get a min cut with decent probability.
  - In contrast, we'd have to choose a random cut about 1/0.008 = 125 times!

Hang on! This "20%" figure just came from running experiments on this particular graph. What about general graphs? Can we prove this?

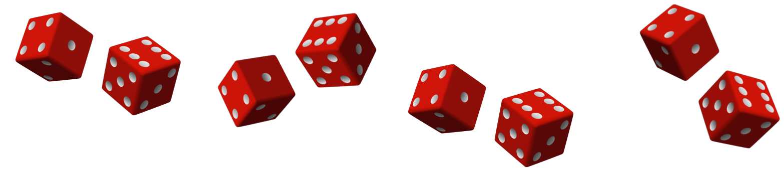
Also, we should be a bit more precise about this "about 5 times" statement.

The plan:

- See that 20% chance of correctness is actually nontrivial.
- Use repetition to boost an algorithm that's correct 20% of the time to an algorithm that's correct 99% of the time.

Questions

To generalize this approach to all graphs



1. What is the probability that Karger's algorithm returns a minimum cut?

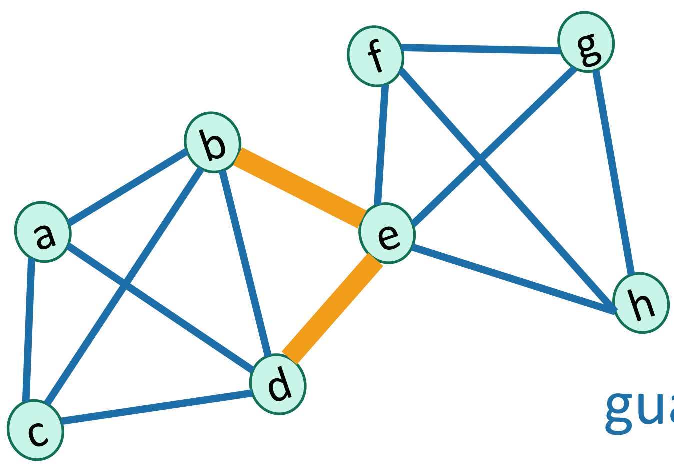
2. How many times should we run Karger's algorithm to "probably" succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

Answer to Question 1

Claim:

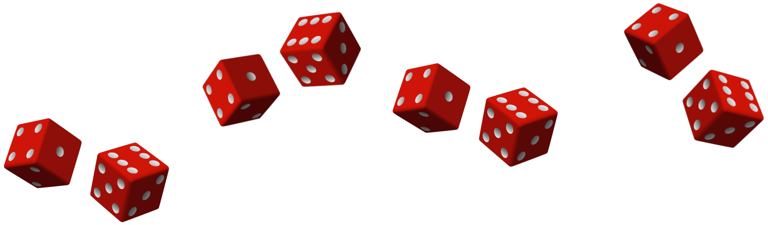
The probability that Karger's algorithm returns a minimum cut is

at least $^1/_{\binom{n}{2}}$



In this case, $^1/_{\binom{8}{2}} = 0.036$, so we are guaranteed to win at least 3.6% of the time.

Answers



1. What is the probability that Karger's algorithm returns a minimum cut?

   According to the claim, at most $\frac{1}{\binom{n}{2}}$

2. How many times should we run Karger's algorithm to "probably" succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

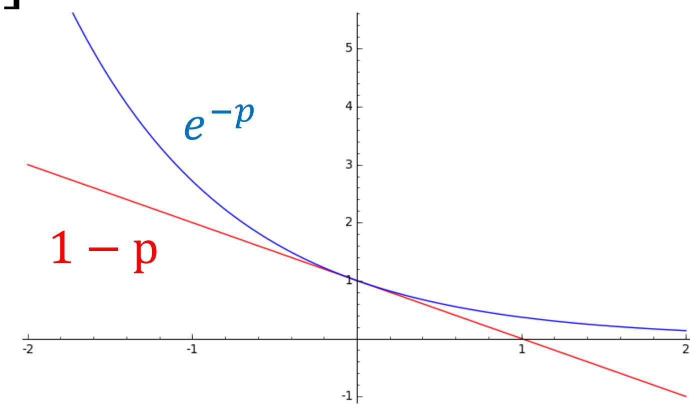
A computation

Punchline: If we repeat **T =** $\binom{n}{2}$ **ln**$(\mathbf{1}/\boldsymbol{\delta})$ times, we win with probability at least $\mathbf{1}-\boldsymbol{\delta}$.

- Suppose :
  - the probability of successfully returning a minimum cut is $\boldsymbol{p} \in [\mathbf{0}, \mathbf{1}]$,
  - we want failure probability at most $\delta \in (0,1)$.

Independent

- $\Pr[\text{ don't return a min cut in T trials }] = (1-p)^T$
- So p = $1/\binom{n}{2}$ by the Claim. Let's choose T = $\binom{n}{2}\ln(1/\delta)$.
- $\Pr[\text{ don't return a min cut in T trials }]$
  - $= (1 \ - p)^T$
  - $\leq (e^{-p})^T$
  - $= e^{-pT}$
  - $= e^{-\ln\left(\frac{1}{\delta}\right)}$
  - $= \delta$



$1-\mathrm{p} \leq e^{-p}$

Theorem

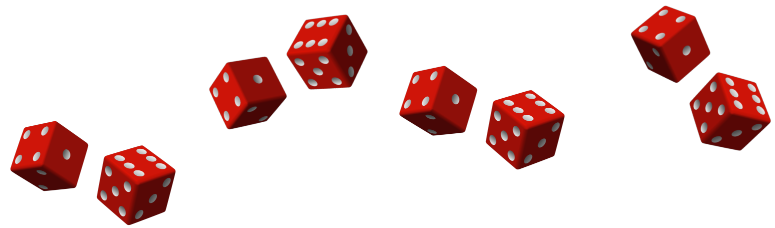
Assuming the claim about $1/\binom{n}{2}$ ...

- Suppose G has n vertices.

- Consider the following algorithm:
  - bestCut = None
  - **for** $\mathrm{t} = 1, \ldots, \binom{n}{2} \ln\left(\frac{1}{\delta}\right)$:
    - candidateCut ← **Karger**(G)
    - **if** candidateCut is smaller than bestCut:
      - bestCut ← candidateCut
  - **return** bestCut

- Then $\Pr[\text{ this doesn't return a min cut }] \leq \delta.$

Answers



1. What is the probability that Karger's algorithm returns a minimum cut?

According to the claim, at most $\frac{1}{\binom{n}{2}}$

2. How many times should we run Karger's algorithm to "probably" succeed?
   - Say, with probability 0.99?
   - Or more generally, probability $1 - \delta$?

$\binom{n}{2} \log\left(\frac{1}{\delta}\right)$ times.

What's the running time?

- $\binom{n}{2}\ln\left(\frac{1}{\delta}\right)$ repetitions, and O(n^2) per repetition.
- So, $O\left(n^2 \cdot \binom{n}{2}\ln\left(\frac{1}{\delta}\right)\right) = \mathrm{O}(\mathrm{n}^4)$

Treating δ as constant.

Theorem

Assuming the claim about $1/\binom{n}{2}$ …

Suppose G has n vertices. Then [repeating Karger's algorithm] finds a min cut in G with probability at least 0.99 in time $O(n^4)$.

What have we learned?

- If we randomly contract edges:
  - It's unlikely that we'll end up with a min cut.
  - But it's not **TOO** unlikely
  - By repeating, we likely will find a min cut.

Here I chose $\delta = 0.01$ just for concreteness.

- Repeating this process:
  - Finds a **global min cut in time O(n^4), with probability 0.99**.
  - We can run a bit faster if we use a **union-find** data structure.

More generally

- Whenever we have a Monte-Carlo algorithm with a small success probability, we can **boost** the success probability by repeating it a bunch and taking the best solution.

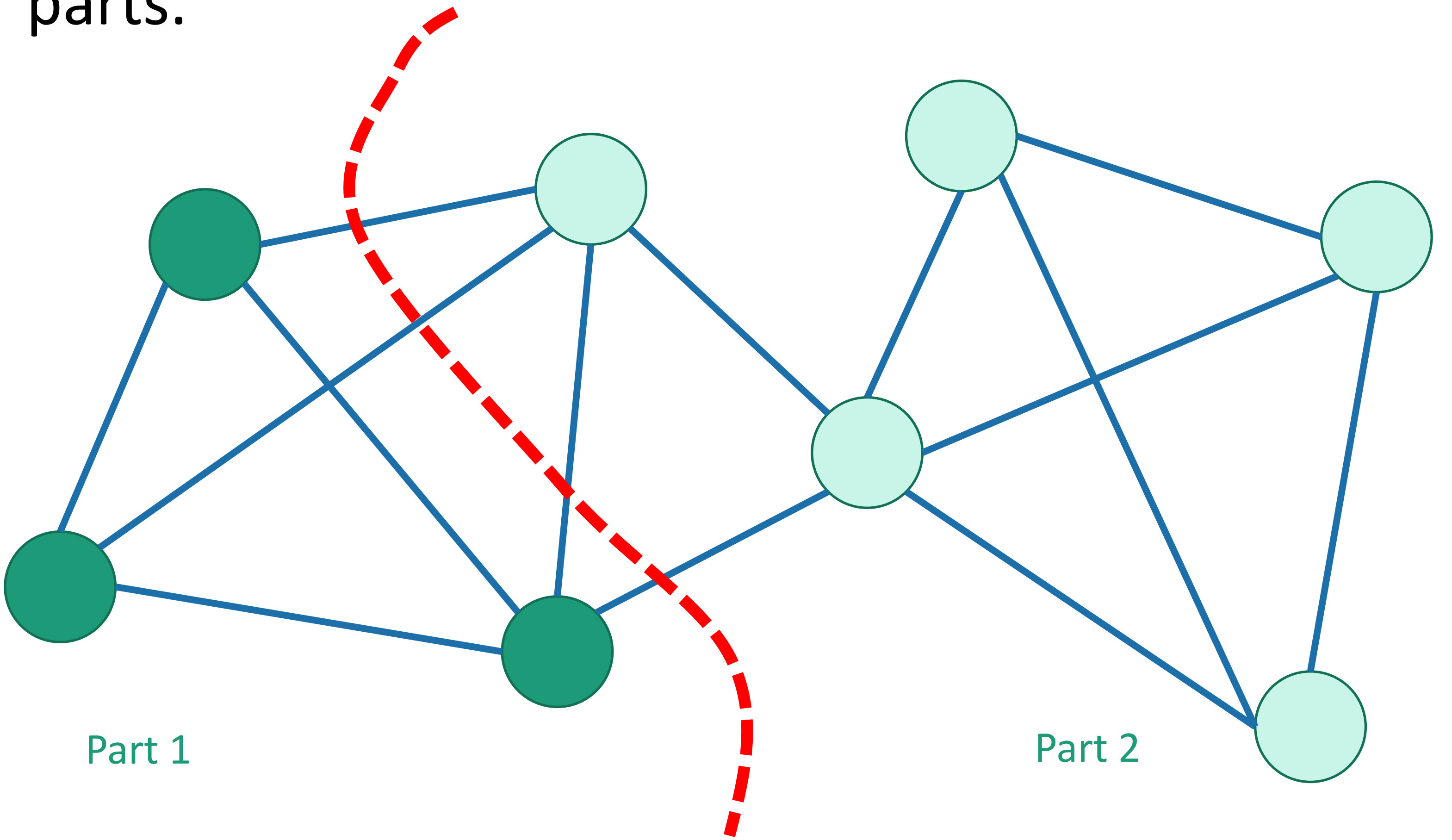
Next

- Minimum s-t cuts
- Maximum s-t flows
- The Ford-Fulkerson Algorithm
  - Finds min cuts and max flows!

Recap

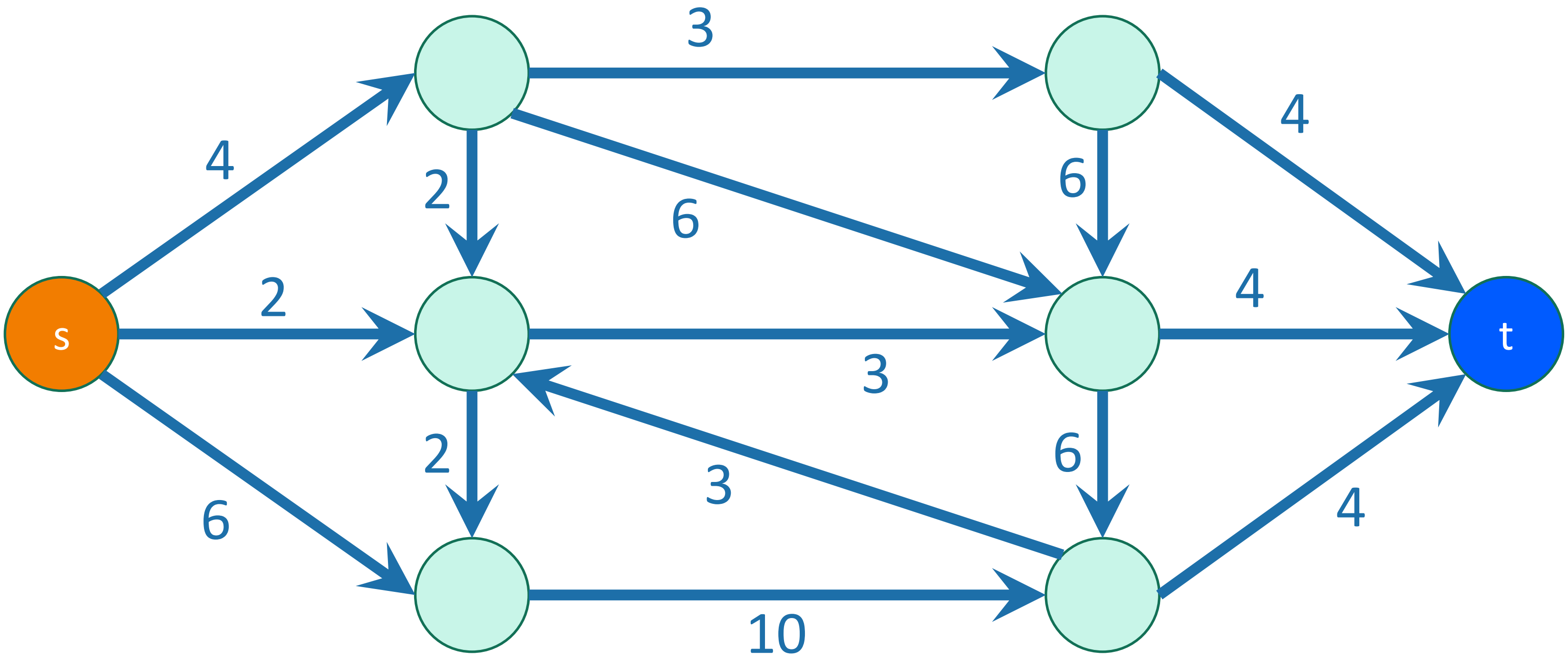
Last time graphs were undirected and unweighted.

- We talked about global min-cuts
- A cut is a partition of the vertices into two nonempty parts.



Graphs with Weights

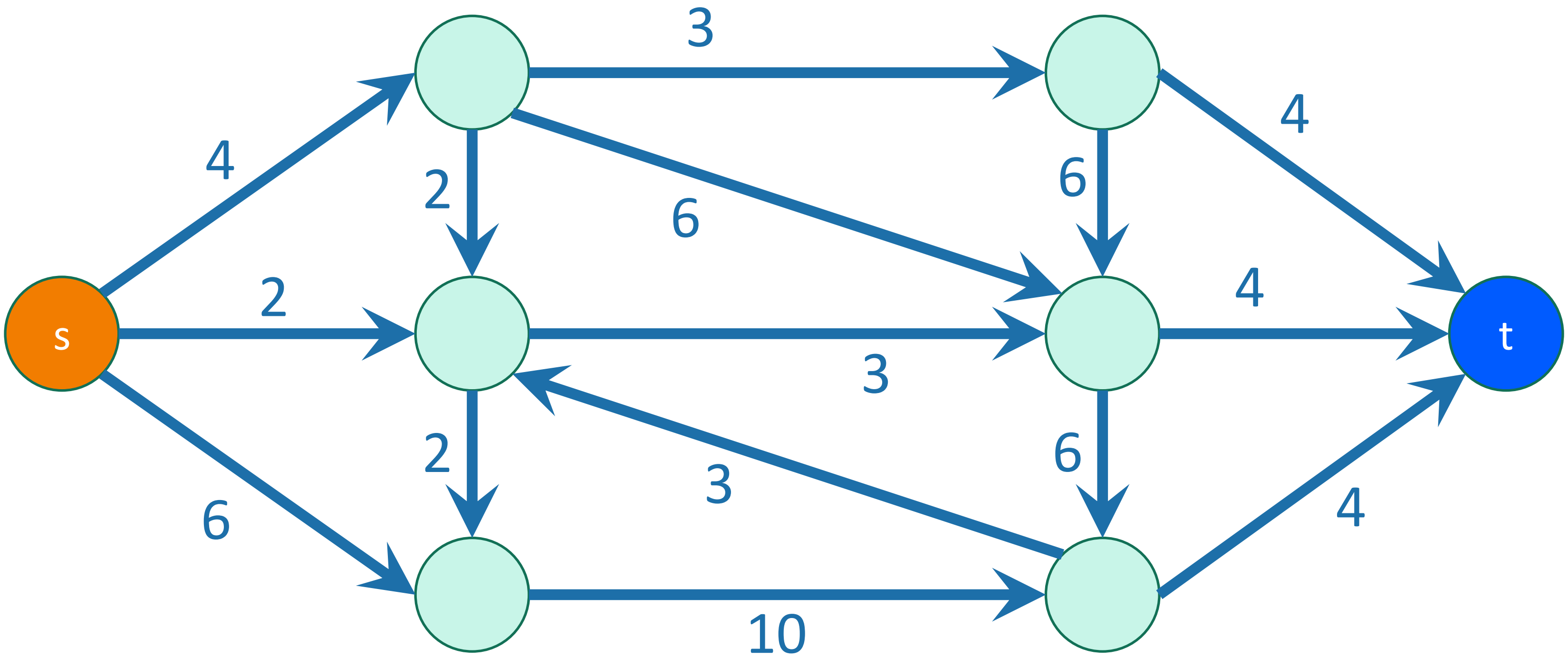
- Graphs are directed and edges have “capacities” (weights)
- We have a special “source” vertex s and “sink” vertex t.
  - s has only outgoing edges*
  - t has only incoming edges*



*at least for this class

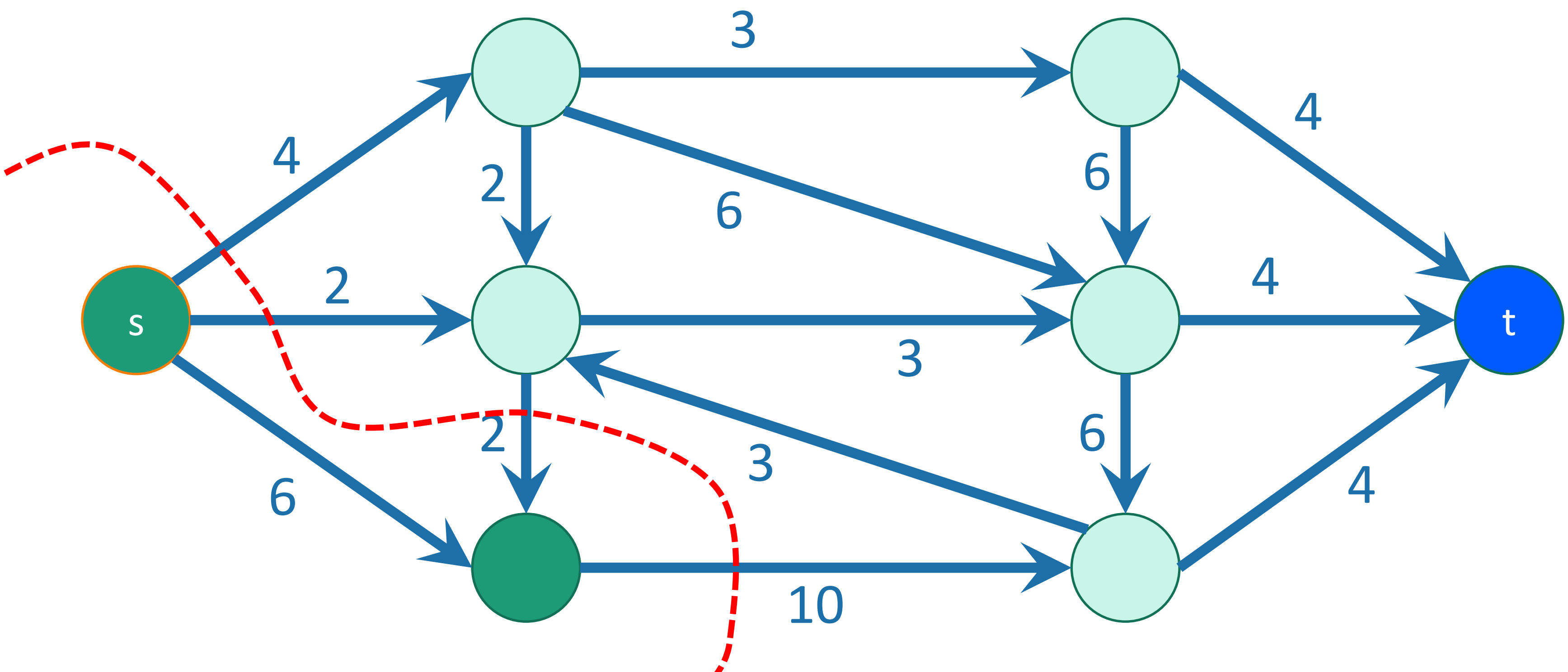
An **s-t cut**

is a cut which separates s from t



An **s-t cut**

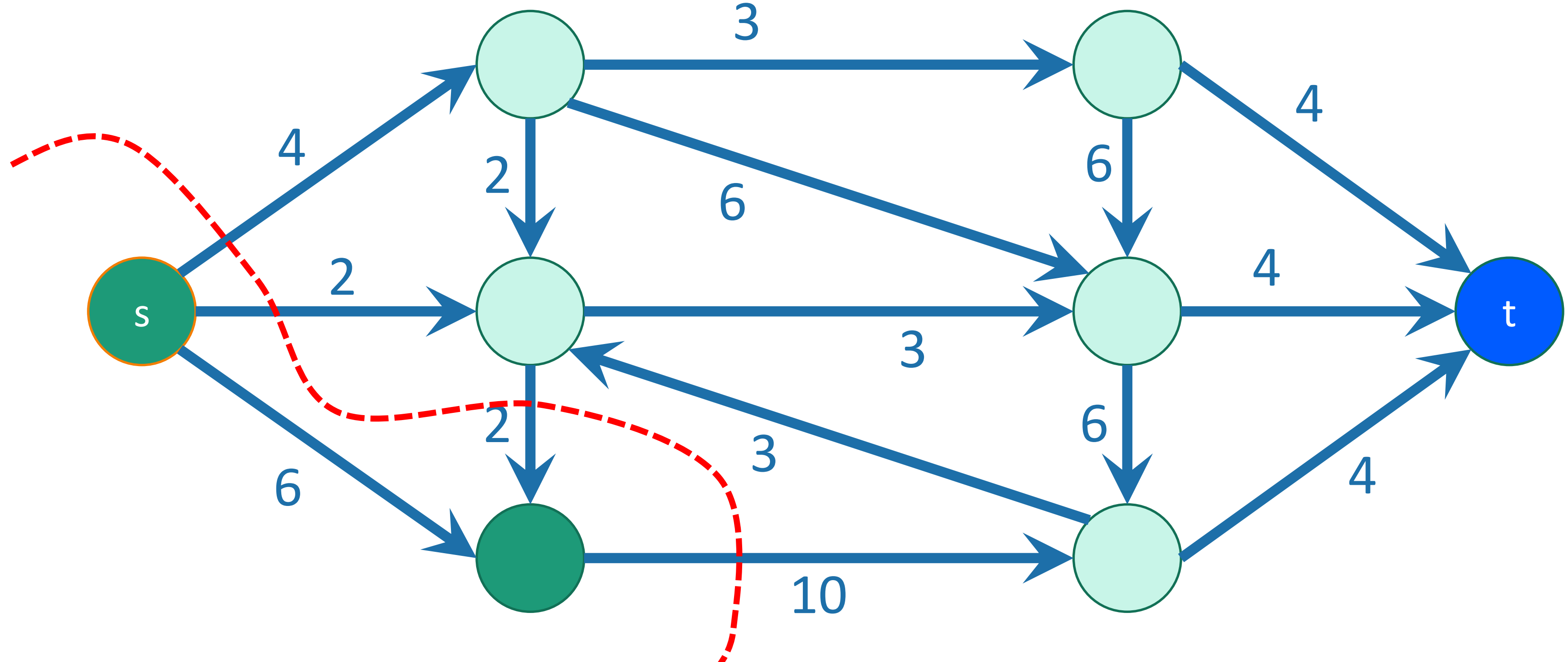
is a cut which separates s from t



An **s-t cut**

is a cut which separates s from t

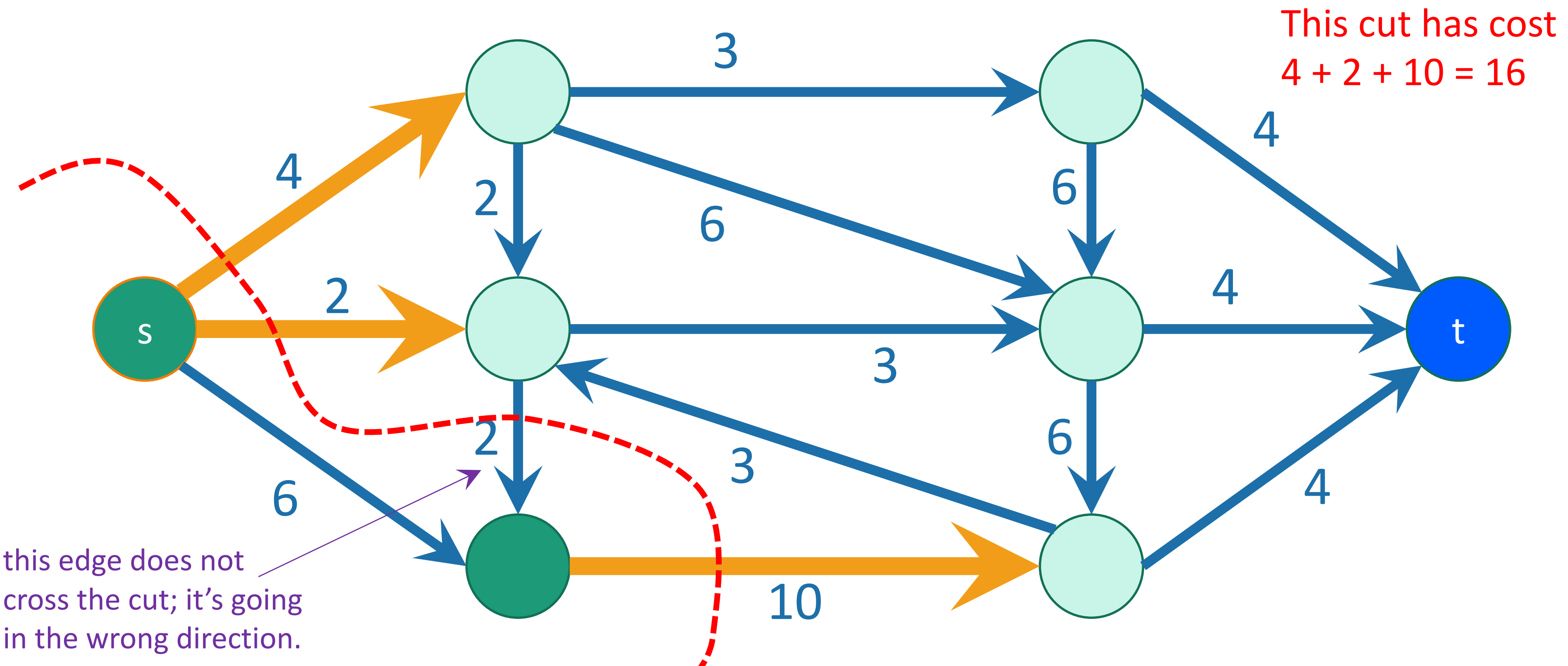
- An edge **crosses the cut** if it goes from s's side to t's side.



An **s-t cut**

is a cut which separates s from t

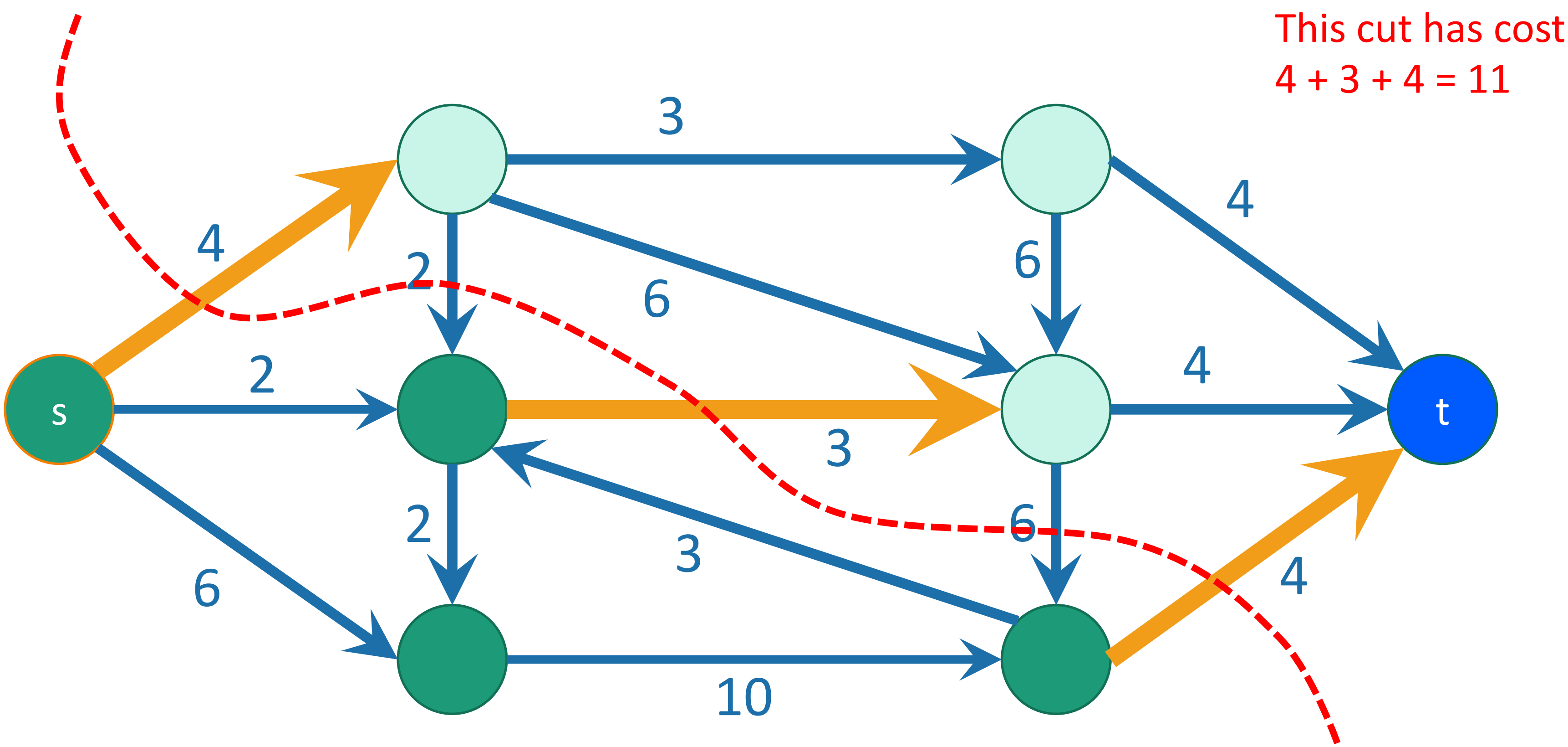
- An edge **crosses the cut** if it goes from s's side to t's side.
- The **cost** (or capacity) of a cut is the sum of the capacities of the edges that cross the cut.



A minimum **s-t cut**

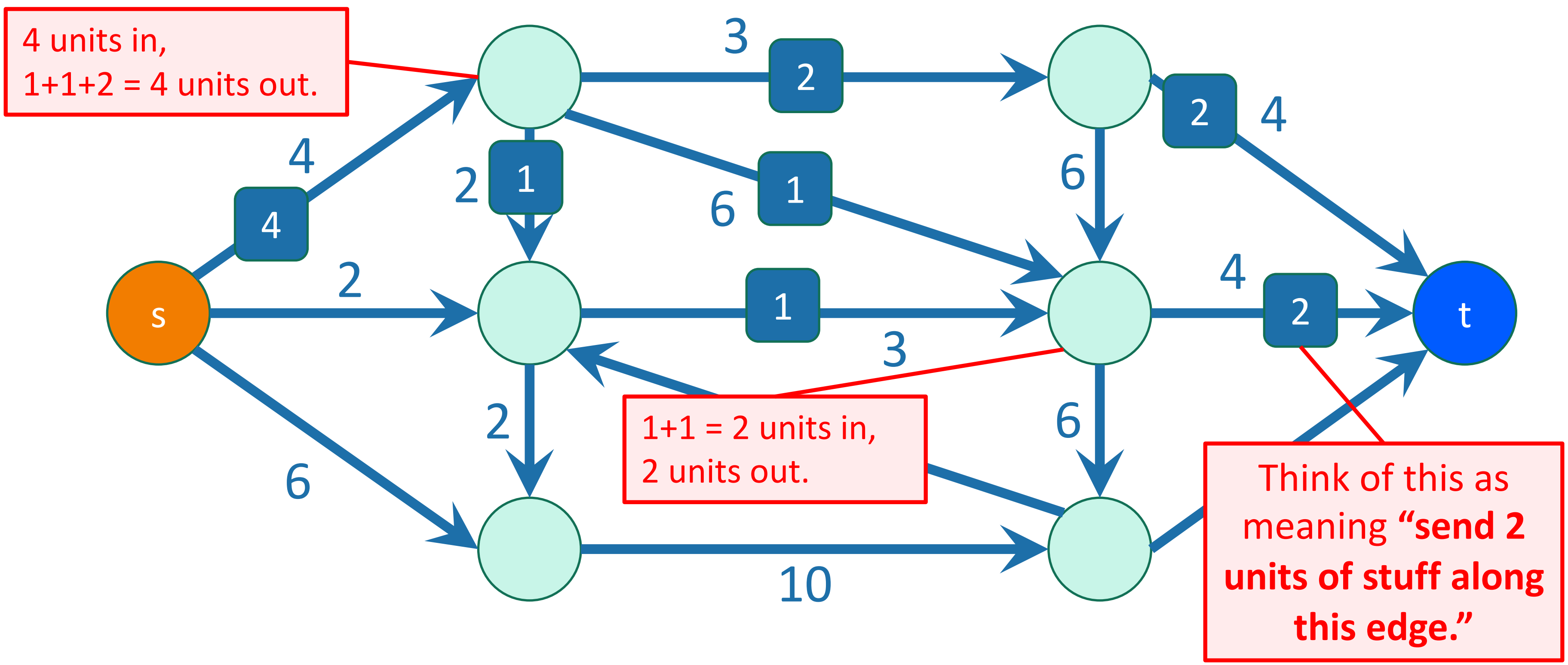
is a cut which separates s from t
with minimum capacity.

- Question: how do we find a minimum 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 less that its capacity.
- At each vertex, the incoming flows must equal the outgoing flows.

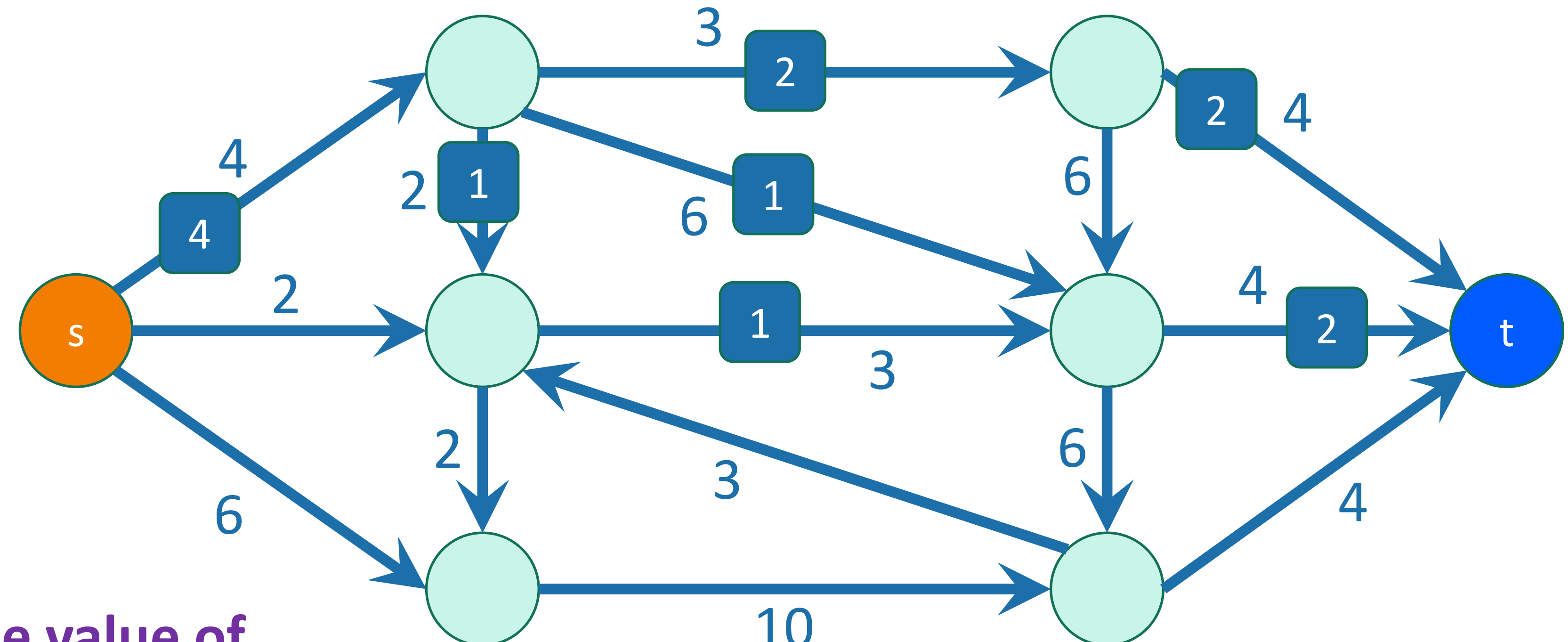


Flows

- 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.

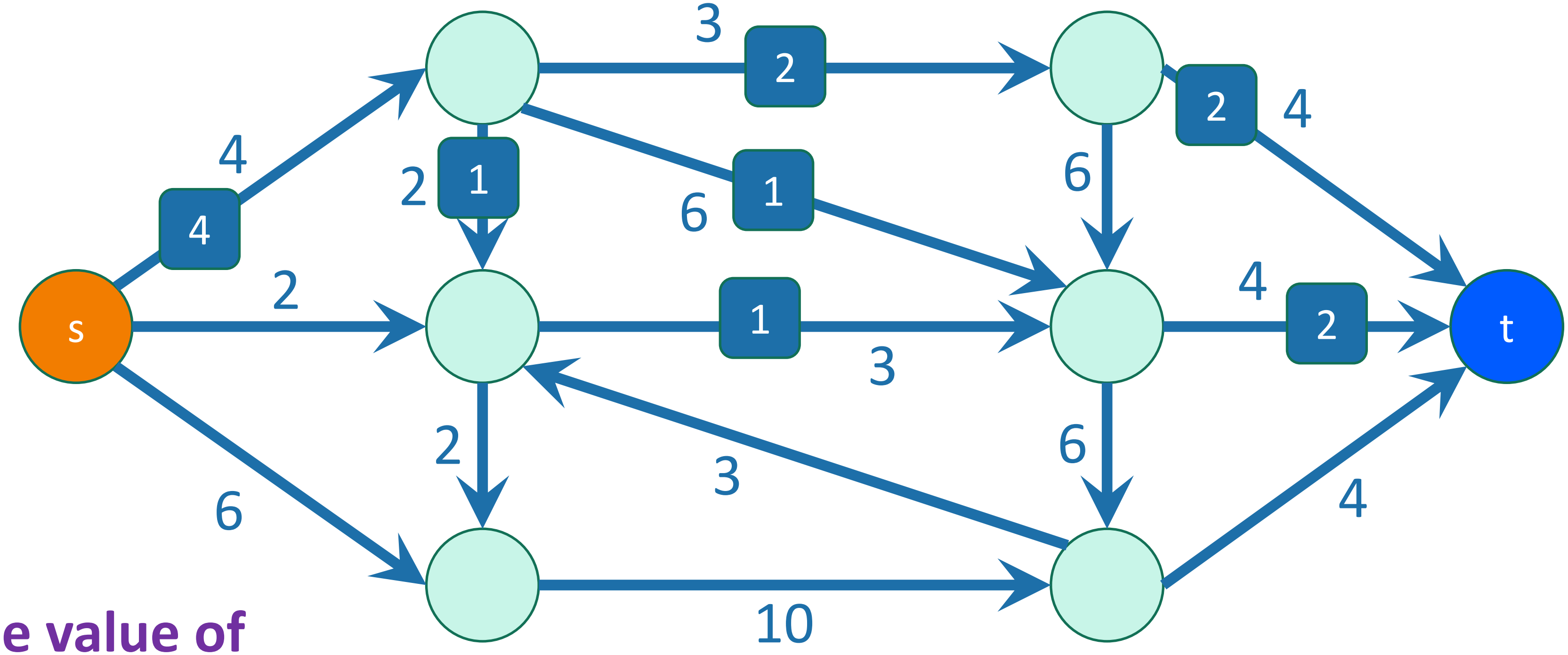


The value of this flow is 4.

A maximum flow

is a flow of maximum value.

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

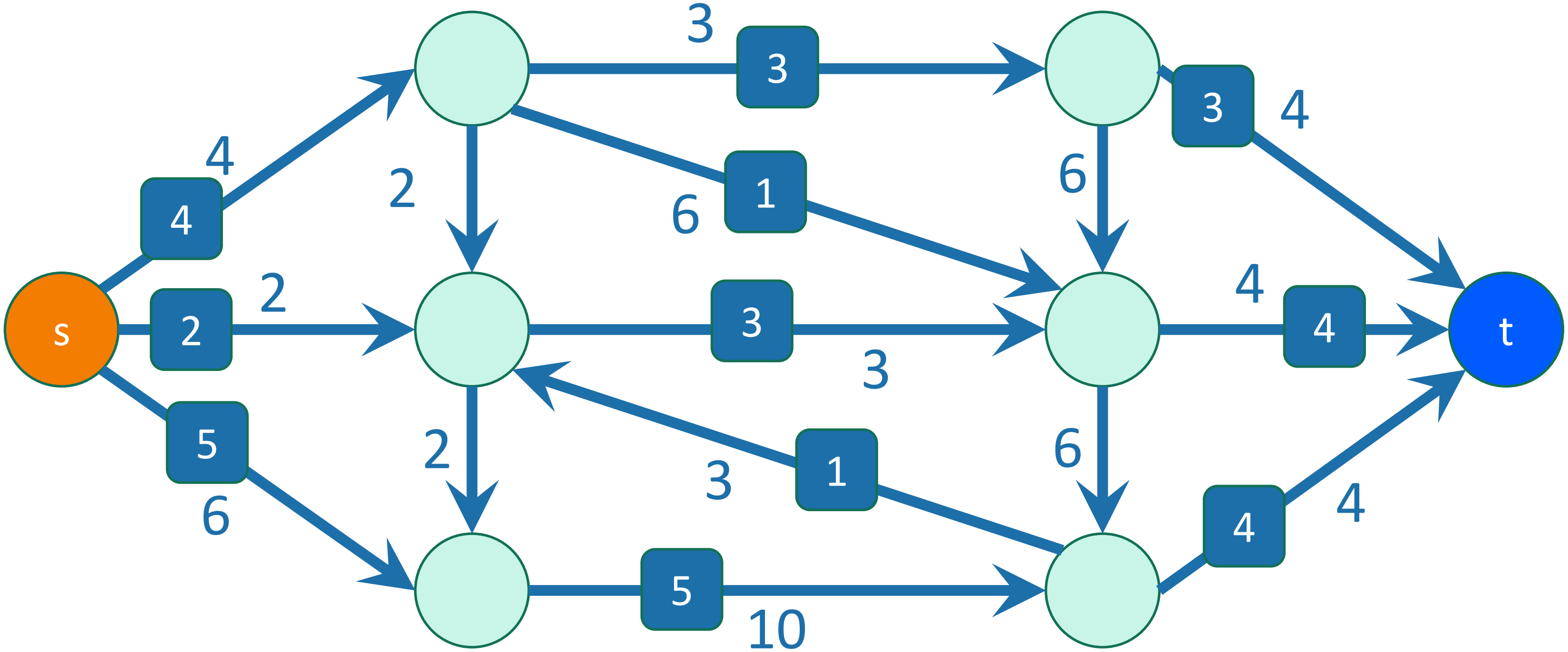


The value of this flow is 4.

A maximum flow

is a flow of maximum value.

- This one is maximal; it has value 11.



Theorem

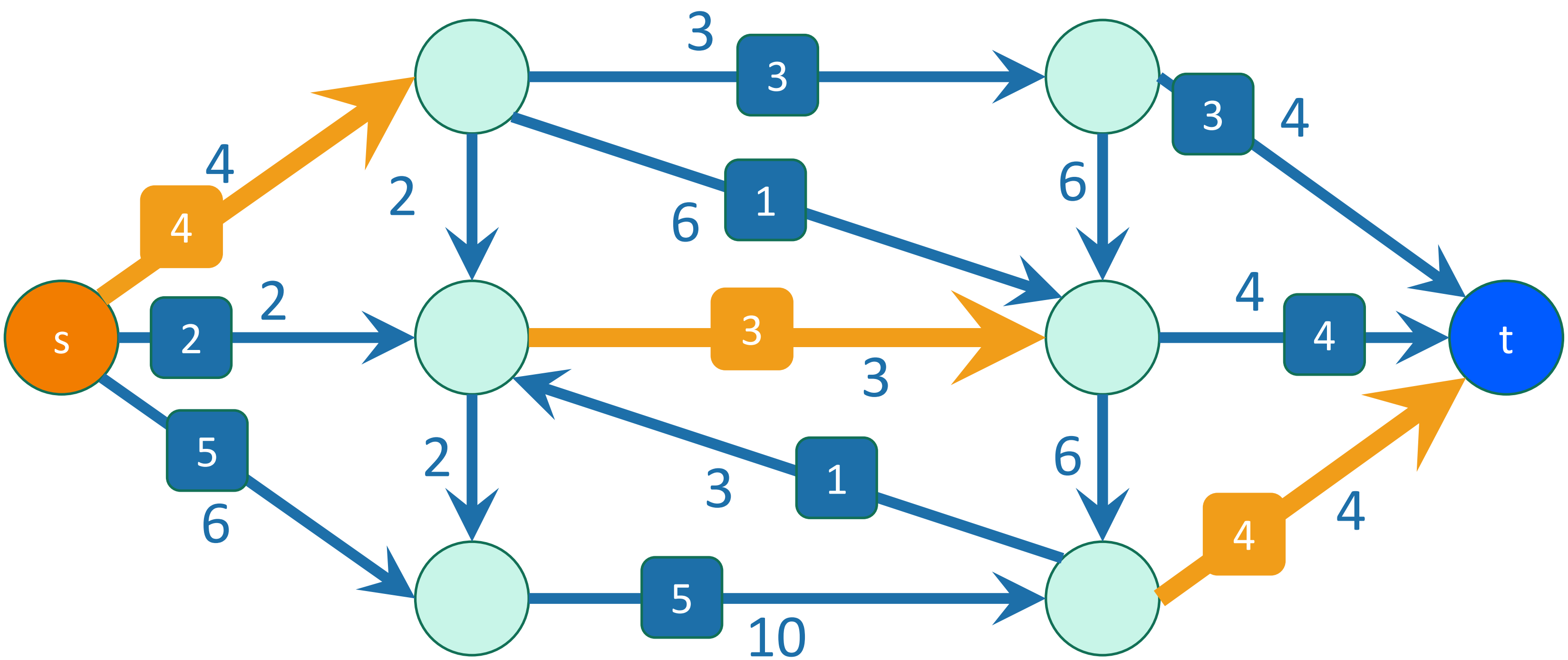
Max-flow min-cut theorem

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.



Proof outline

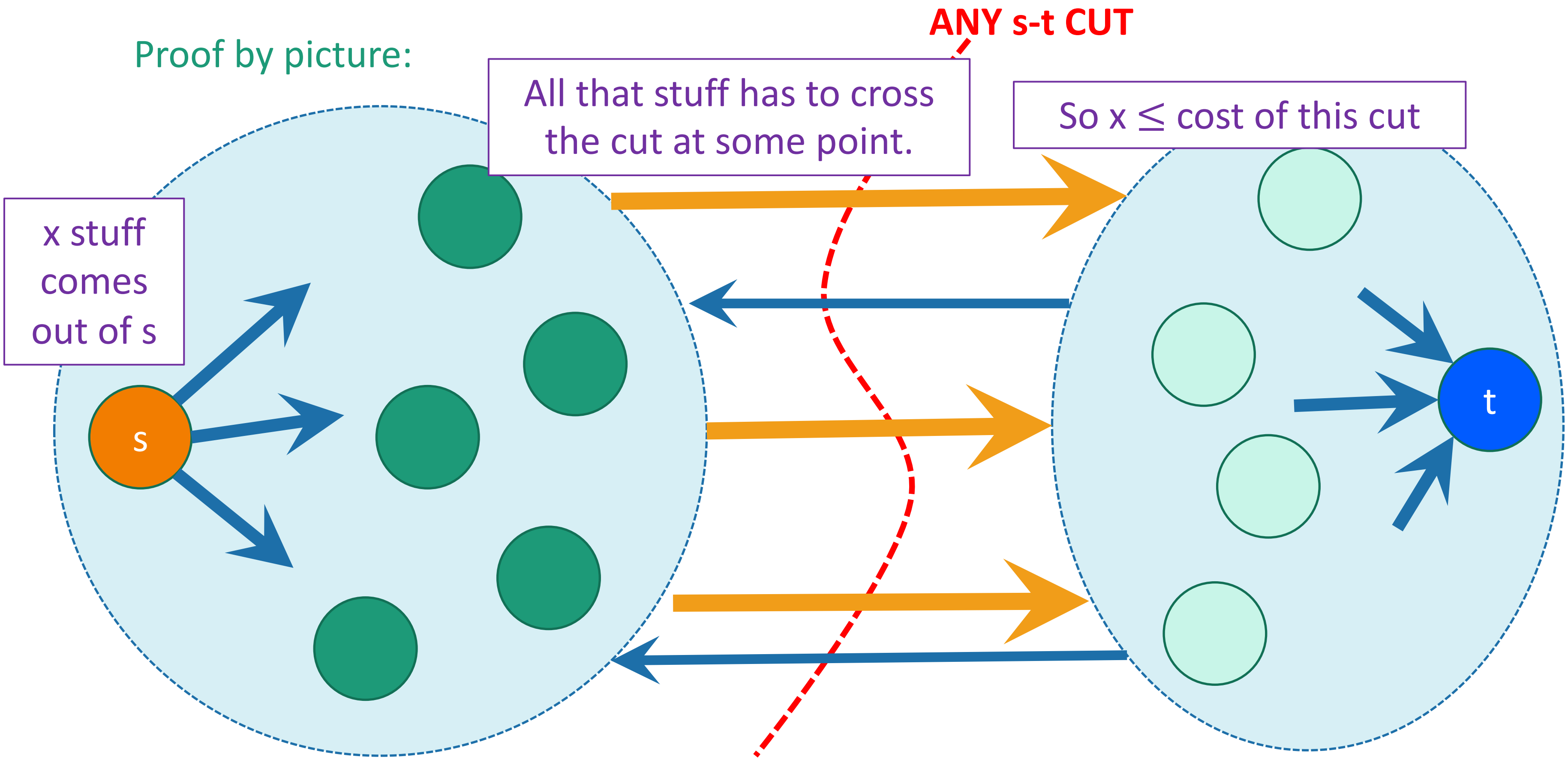
- Lemma 1: max flow $\leq$ min cut.
  - Proof-by-picture
- Lemma 2: max flow $\geq$ min cut.
  - Proof-by-algorithm, using a “Residual graph” G_f
  - Sub-Lemma: t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.
    - $\Leftarrow$ first we do this direction:
    - Claim: If there is a path from s to t in G_f, then we can increase the flow in *G*.
    - Hence we couldn’t have started with a max flow.
    - $\Rightarrow$ for this direction, proof-by-picture again.

This claim actually gives us an algorithm: Find paths from s to t in G_f and keep increasing the flow until you can’t anymore.

Proof of Min-Cut Max-Flow Thm

- **Lemma 1:**
  - For ANY s-t flow and ANY s-t cut, the value of the flow is at most the cost of the cut.
  - Hence max flow ≤ min cut.

Proof by picture:



Proof of Min-Cut Max-Flow Thm

- **Lemma 1:**
  - For ANY s-t flow and ANY s-t cut, the value of the flow is at most the cost of the cut.
  - Hence max flow $\leq$ min cut.

Proof of Min-Cut Max-Flow Thm

- **Lemma 1:**
  - For ANY s-t flow and ANY s-t cut, the value of the flow is at most the cost of the cut.
  - Hence max flow $\leq$ min cut.

- The theorem is stronger:
  - max flow $=$ min cut
  - Need to show max flow $\geq$ min cut.
  - Next: Proof by algorithm!

Ford-Fulkerson algorithm

- Usually we state the algorithm first and then prove that it works.
- Today we're going to just start with the proof, and this will inspire the algorithm.

Outline of algorithm:

- Start with zero flow
- 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.

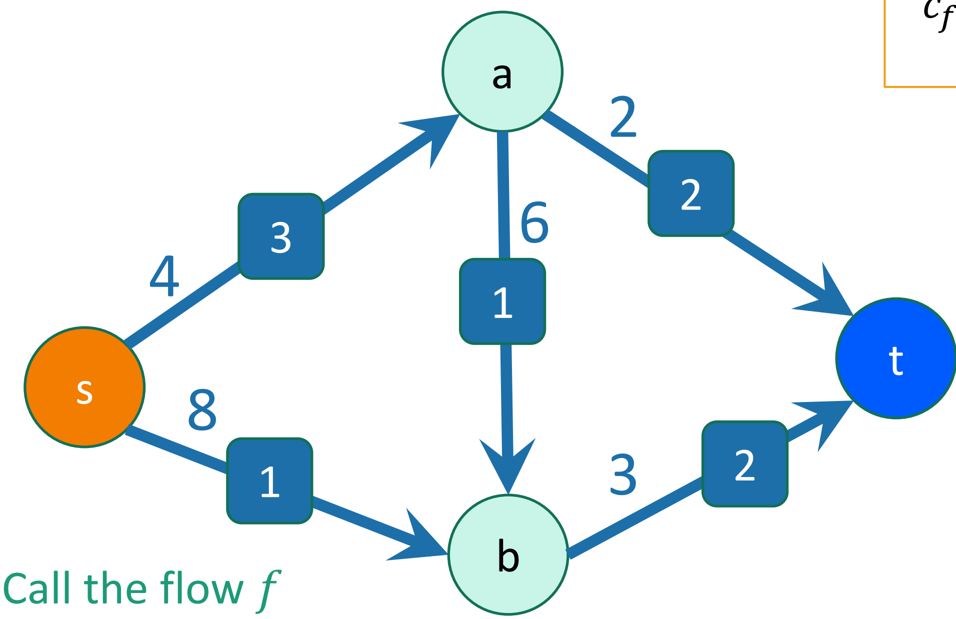
Assume for today that we don't have edges like this, although it's not necessary.

Tool: Residual networks

Say we have a flow

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if \ \ (u,v) \in E \\ f(v,u) & if \ (v,u) \in E \\ 0 & else \end{cases}$$

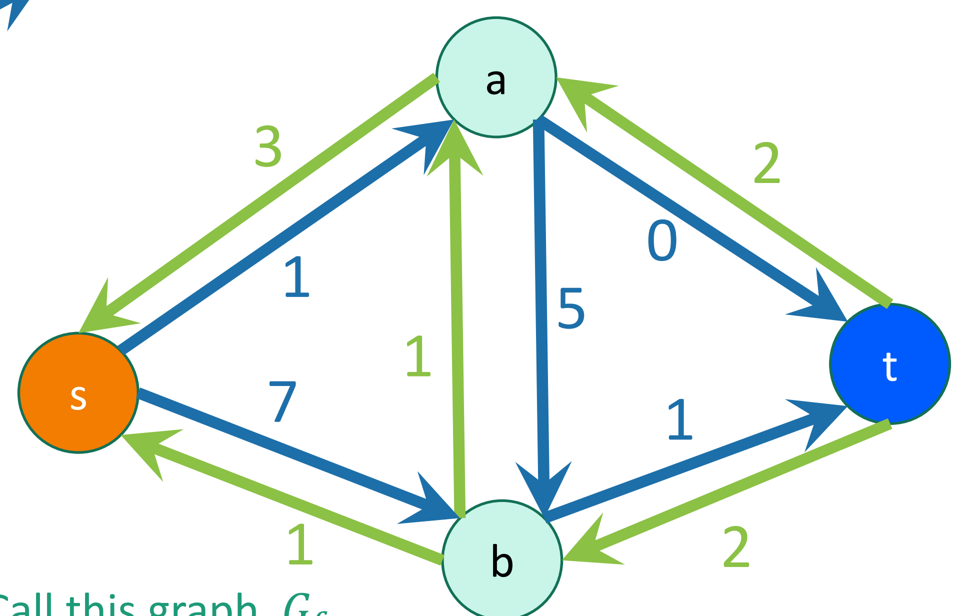
- $f(u,v)$ is the flow on edge (u,v).
- $c(u,v)$ is the capacity on edge (u,v)



Call the flow f

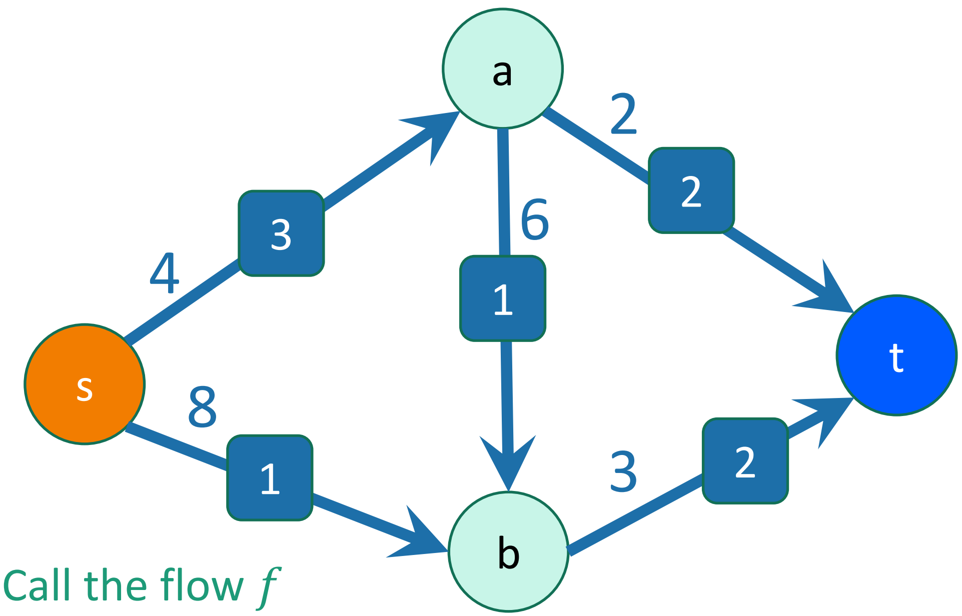
Call the graph G

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



Call this graph G_f

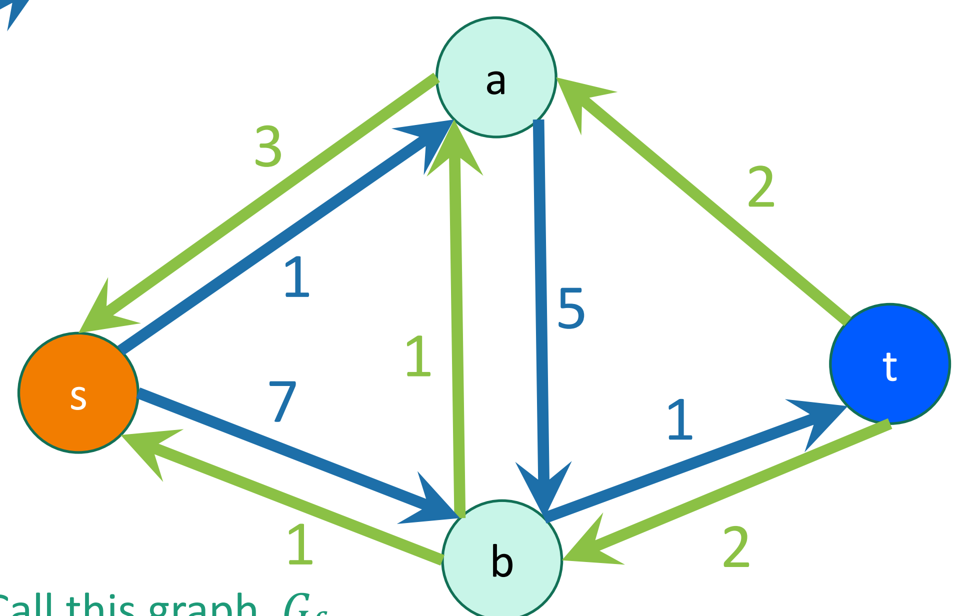
Tool: Residual networks
Say we have a flow



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:



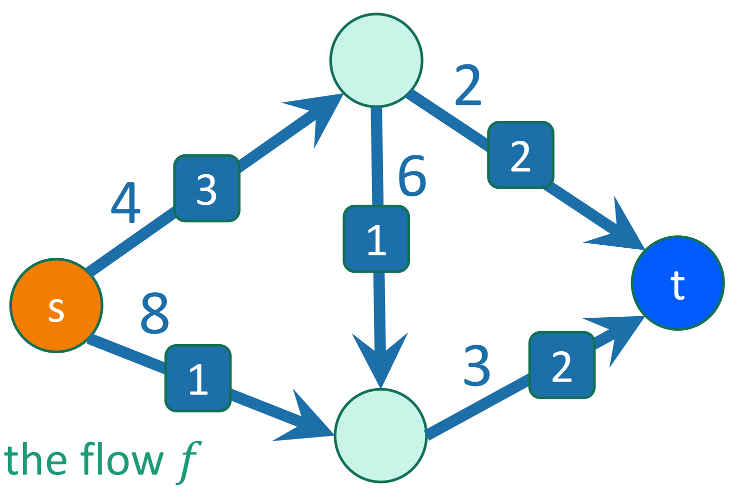
Call this graph G_f

Why look at residual networks?

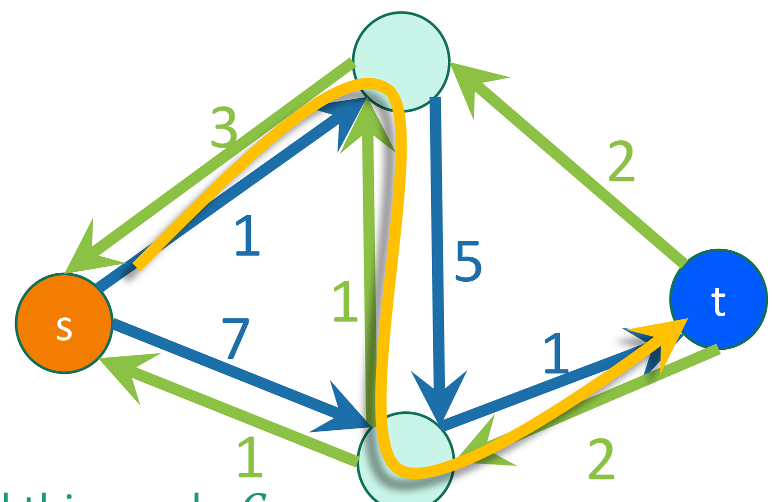
Lemma:

- t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.

Example: **s is reachable from t in this example, so not a max flow.**



Call the flow f
Call the graph G



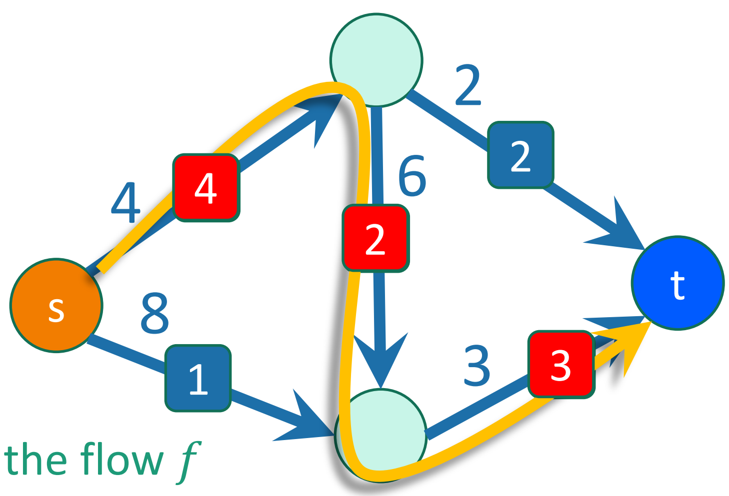
Call this graph G_f

Why look at residual networks?

Lemma:

- t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.

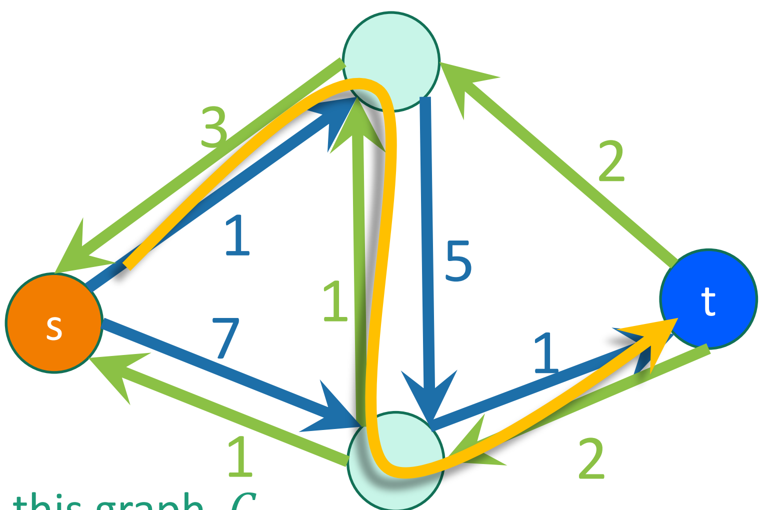
To see that this flow is not maximal, notice that we can improve it by sending one more unit more stuff along this path:



Call the flow f
Call the graph G

Example: **s is reachable from t in this example, so not a max flow.**

Now update the residual graph…



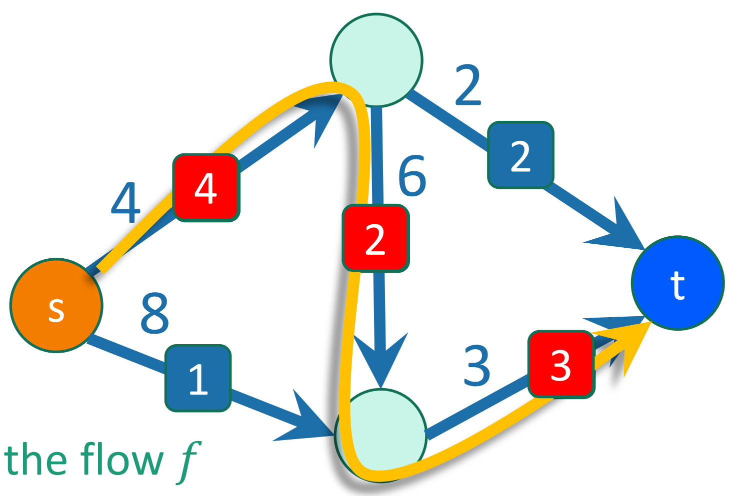
Call this graph G_f

Why look at residual networks?

Lemma:

- t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.

To see that this flow is not maximal, notice that we can improve it by sending one more unit more stuff along this path:

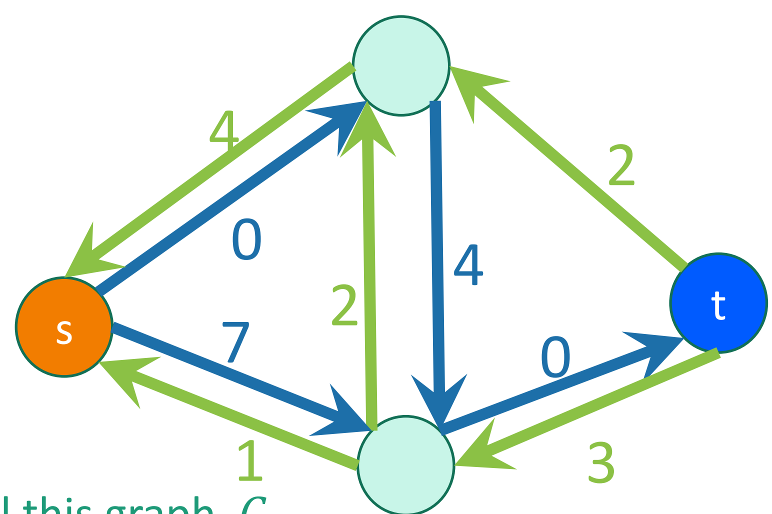


Call the flow f

Call the graph G

Example:

Now we get this residual graph:



Call this graph G_f

Why look at residual networks?

Lemma:

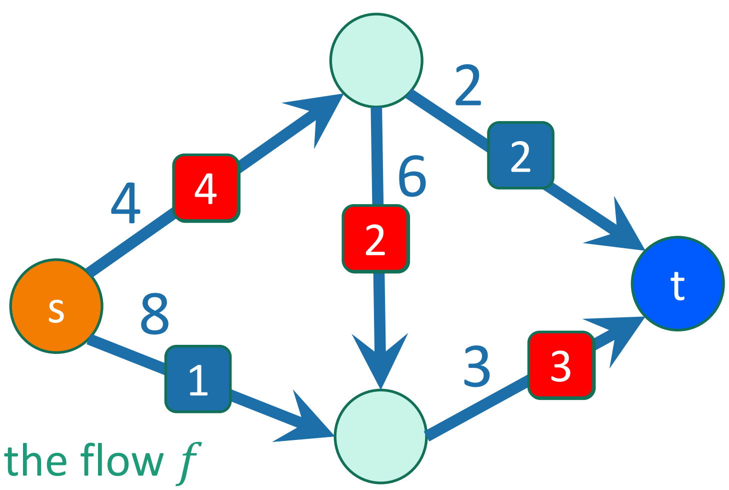
- t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.

Example:

Now we get this residual graph:

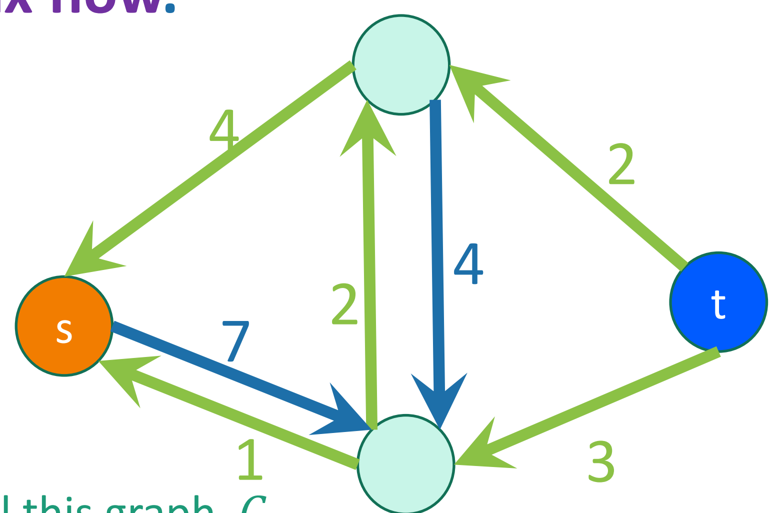
Now we can't reach t from s.

So the lemma says that f is a max flow.



Call the flow f

Call the graph G



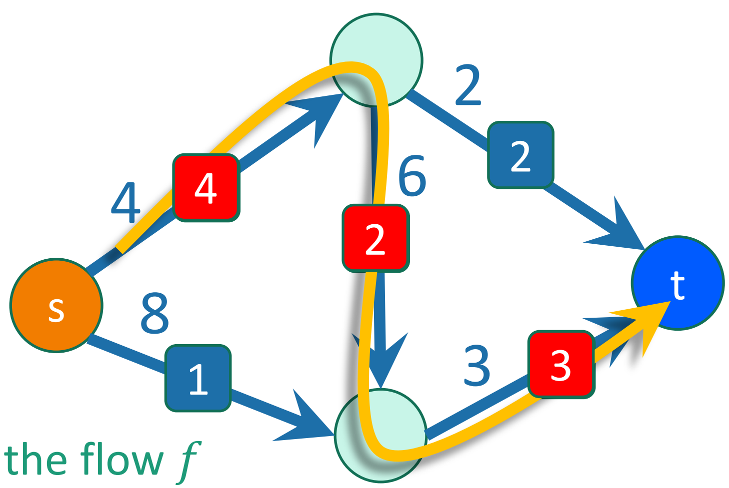
Call this graph G_f

Lemma:

t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow.

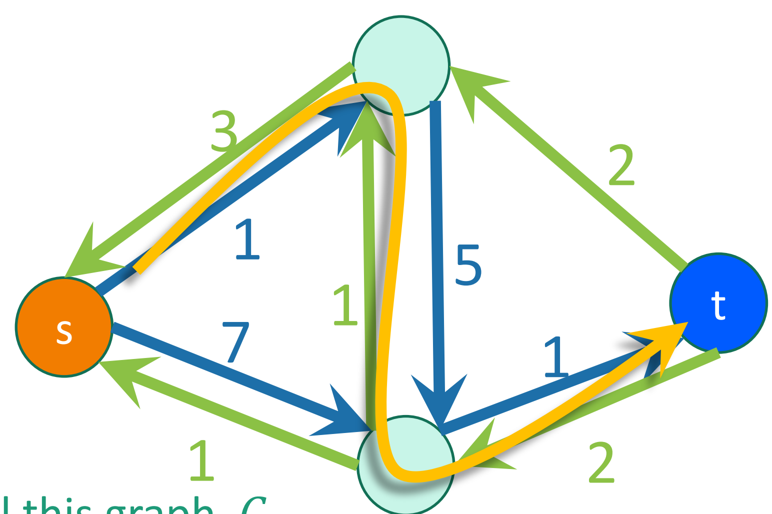
- Suppose there is a path from s to t in G_f.
  - This is called an augmenting path.
- **Claim:** if there is an augmenting path, we can increase the flow along that path.
- So do that and update the flow.
- This results in a bigger flow
  - so we can't have started with a max flow.

we will come back to this in a second.



Call the flow f

Call the graph G

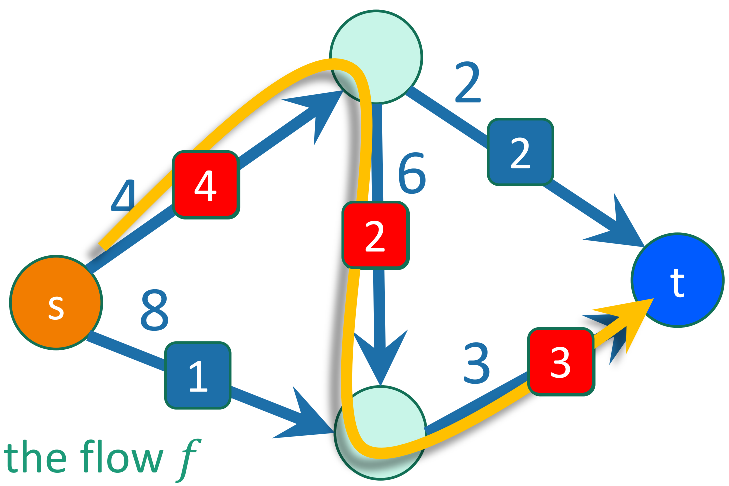


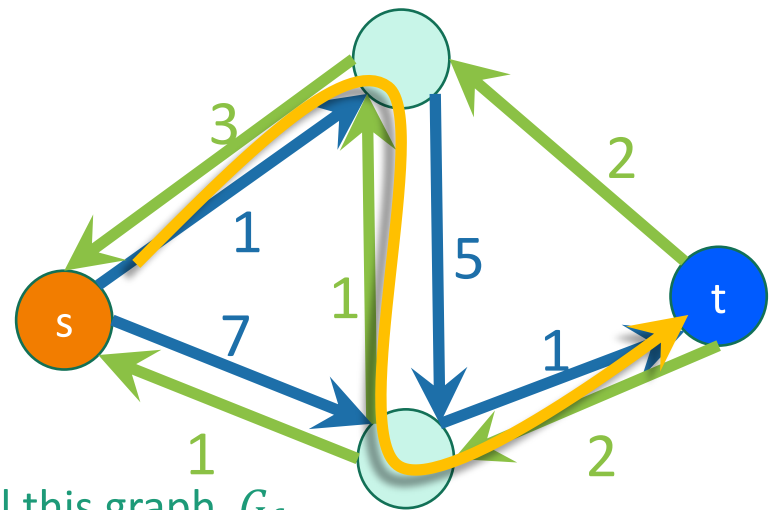
Call this graph G_f

claim:

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

- In the situation we just saw, this is pretty obvious.





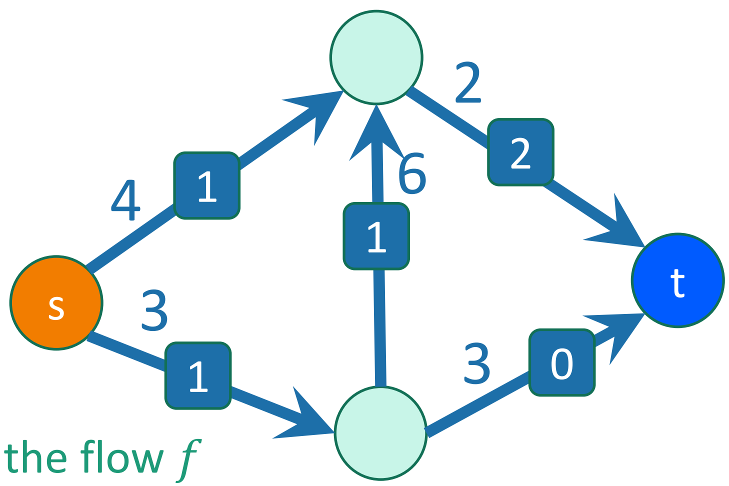
- Every edge on the path in G_f was a **forward edge**, so increase the flow on all the edges.

aka, an edge indicating how much stuff can still go through

claim:

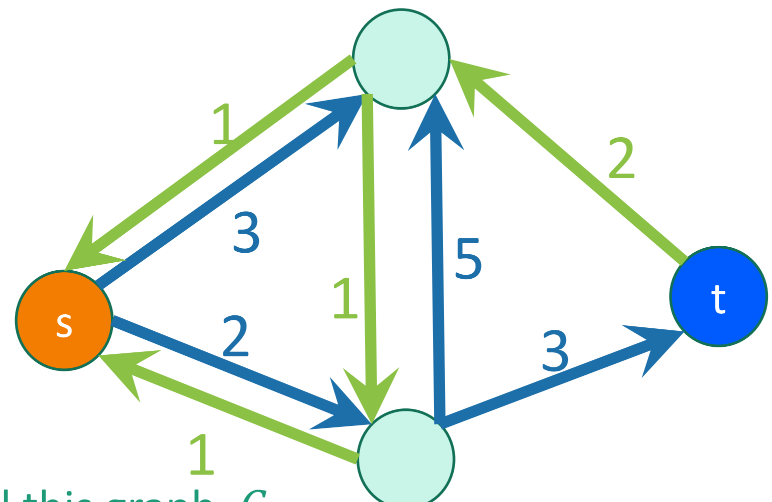
if there is an augmenting path, we can increase the flow along that path.

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



Call the flow f

Call the graph G



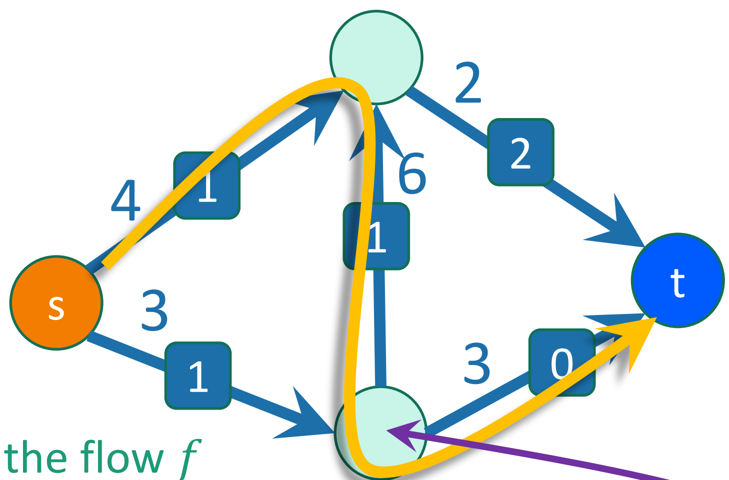
Call this graph G_f

I changed some of the weights and edge directions.

claim:

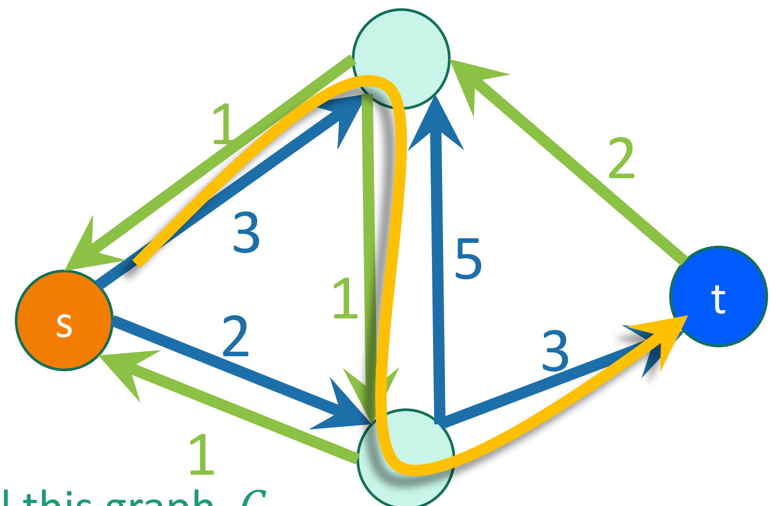
if there is an augmenting path, we can increase the flow along that path.

- But maybe there are **backward edges** 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

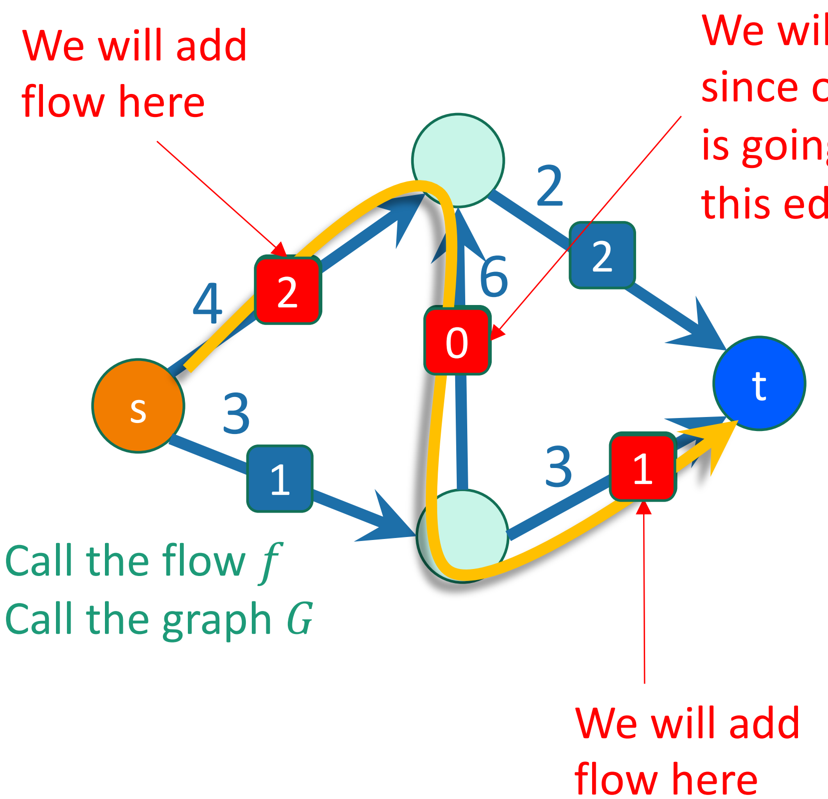
Now we should NOT increase the flow at all the edges along the path!

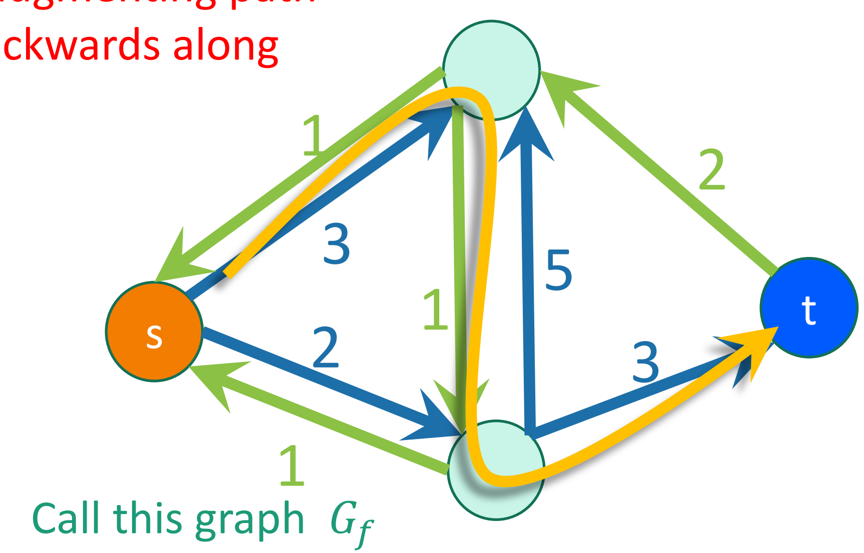
- For example, that will mess up the conservation of stuff at this vertex.

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:



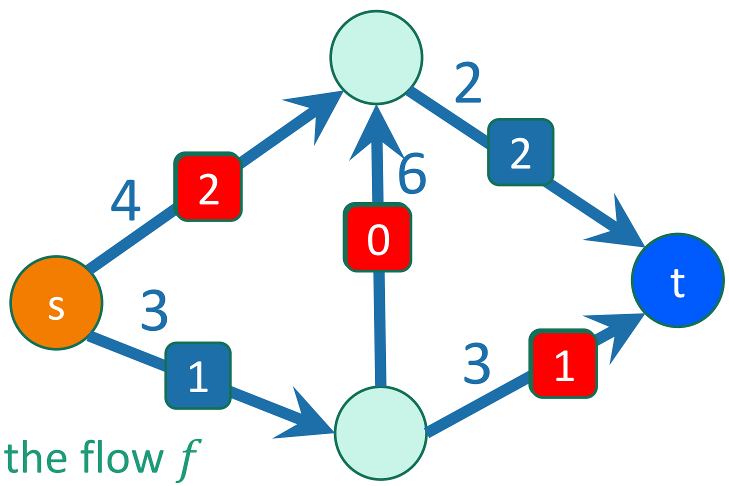


claim:

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

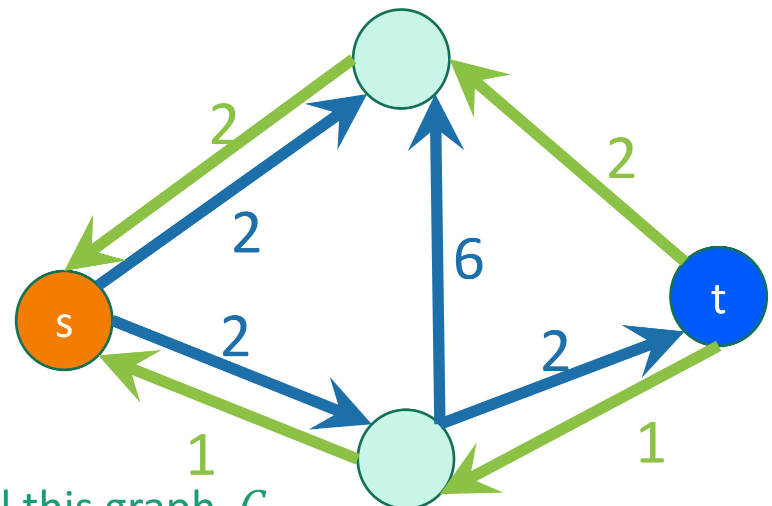
- In this case we do something a bit different:

Then we'll update the residual graph:



Call the flow f

Call the graph G

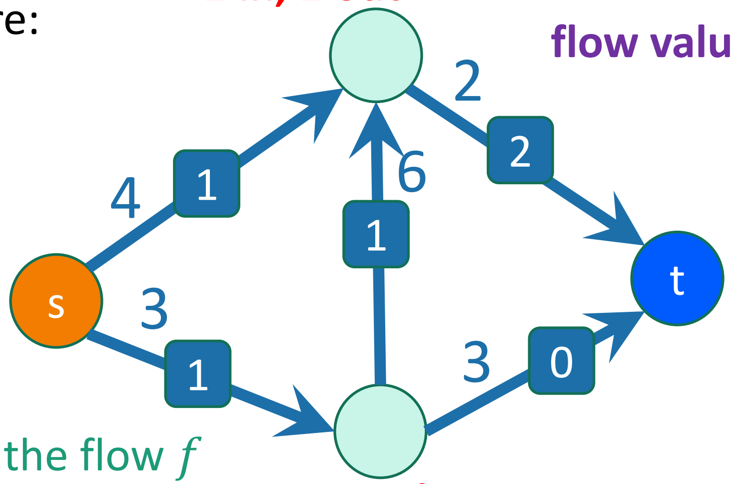


Call this graph G_f

Before:

2 in, 2 out

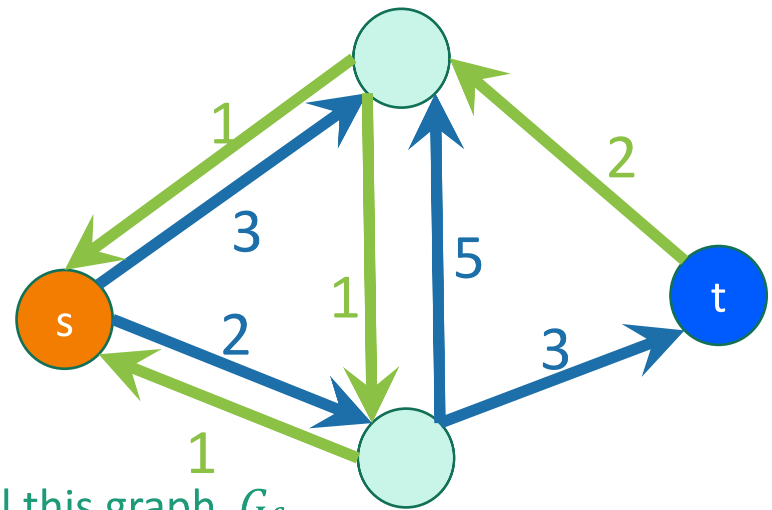
flow value is 2



1 in, 1 out

Call the flow f

Call the graph G

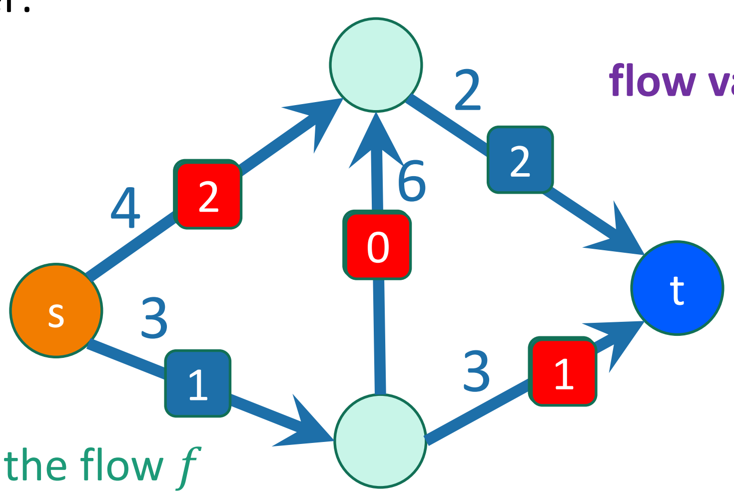


Call this graph G_f

After:

2 in, 2 out

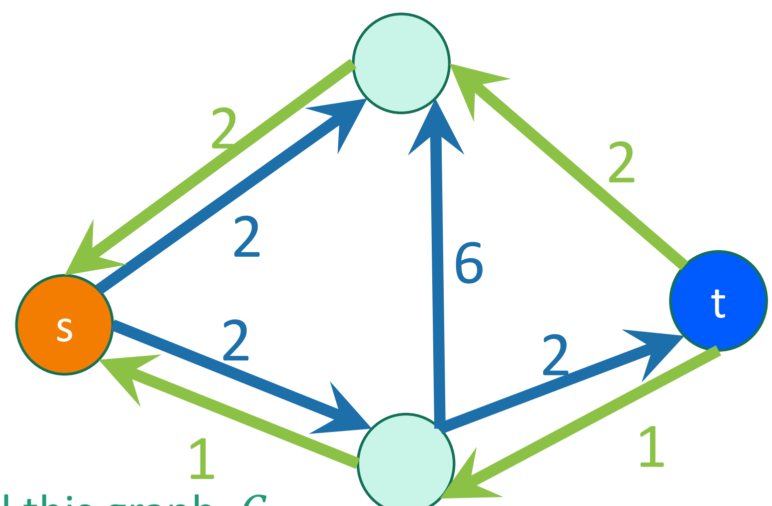
flow value is 3



1 in, 1 out

Call the flow f

Call the graph G



Call this graph G_f

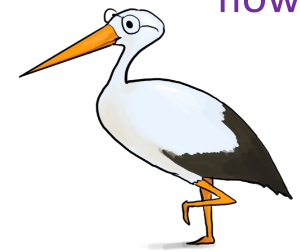
Still a legit flow, but with a bigger value!

claim:

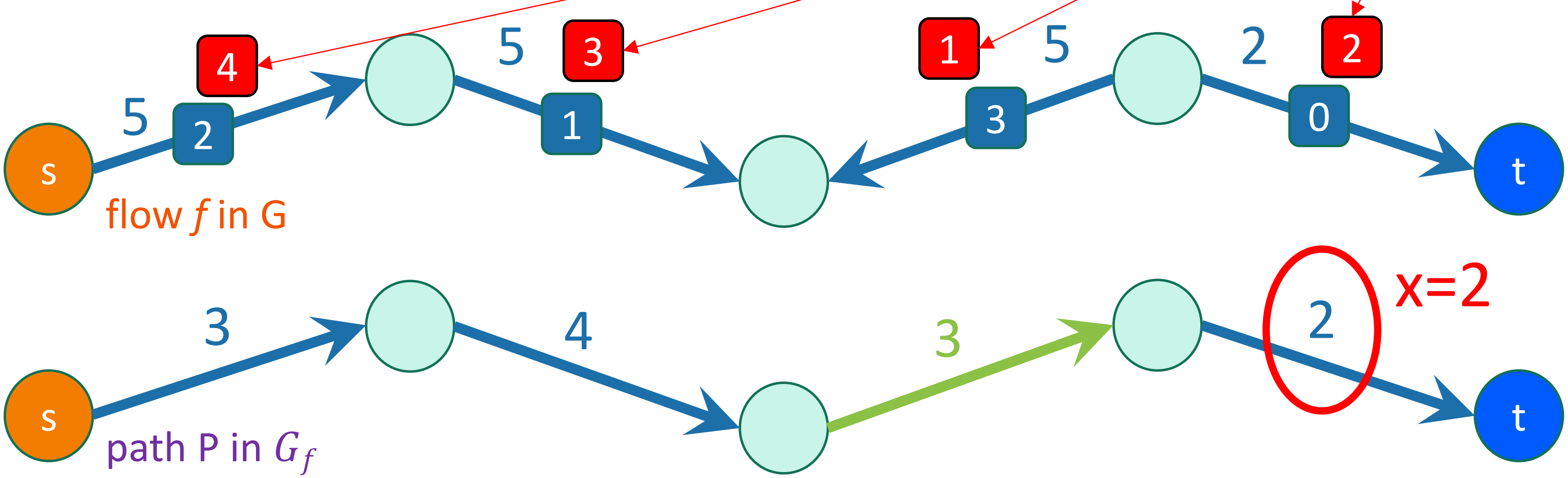
if there is an augmenting path, we can increase the flow along that path.

- 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!



This is f'



That proves the **claim**

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

We've proved:

- t is not reachable from s in G_f $\Leftrightarrow$ f is a max flow

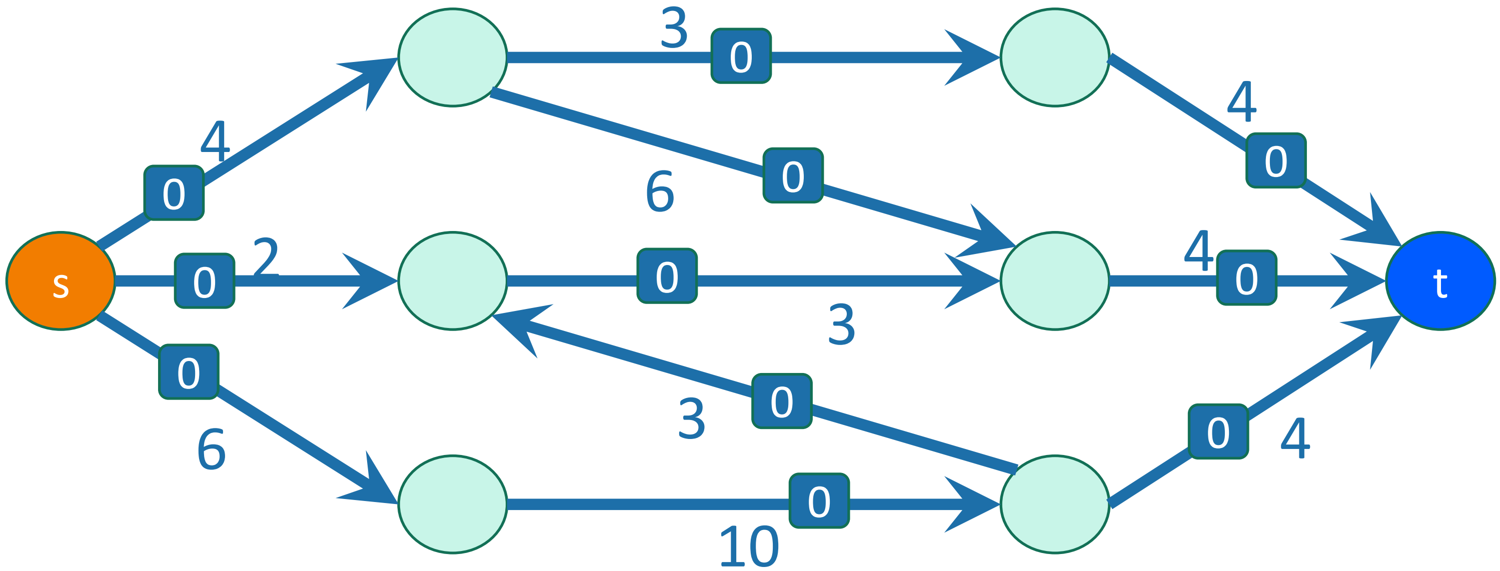
- This is inspires an **algorithm**:
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f // eg, use BFS
    - $f \leftarrow$ **increaseFlow**(*P*,*f*)
    - update G_f
  - **return** f

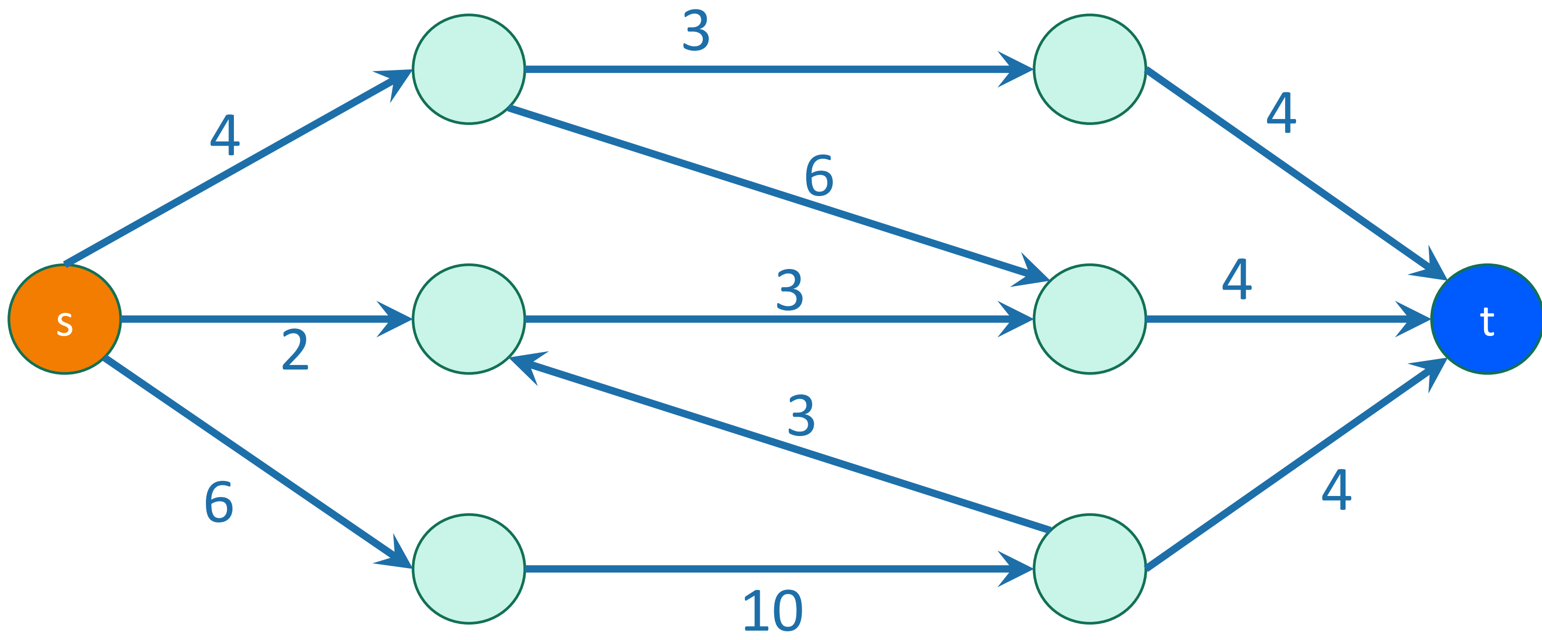
How do we choose which paths to use?

- The analysis we did still works no matter how we choose the paths.
  - That is, the algorithm will be **correct** if it terminates.
- **However, the algorithm may not be efficient!!!**
  - May take a long time to terminate

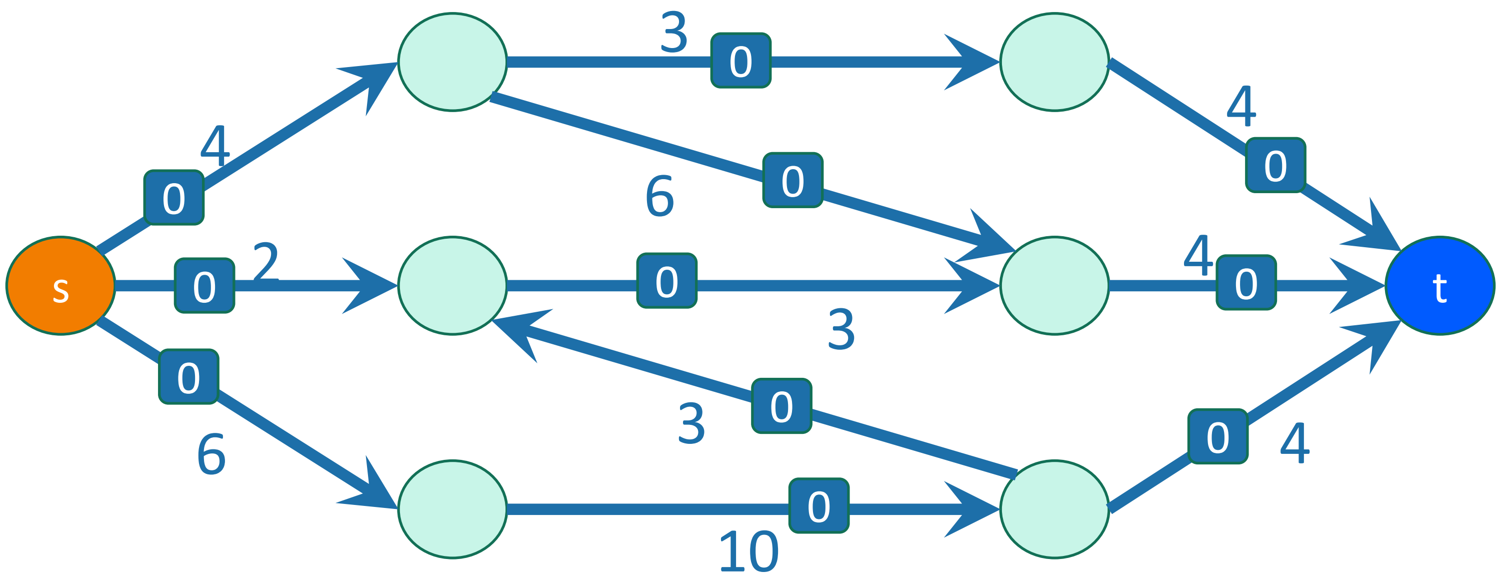
- We need to be careful with our path selection to make sure the algorithm terminates quickly.
  - Using BFS leads to the **Edmonds-Karp algorithm.**

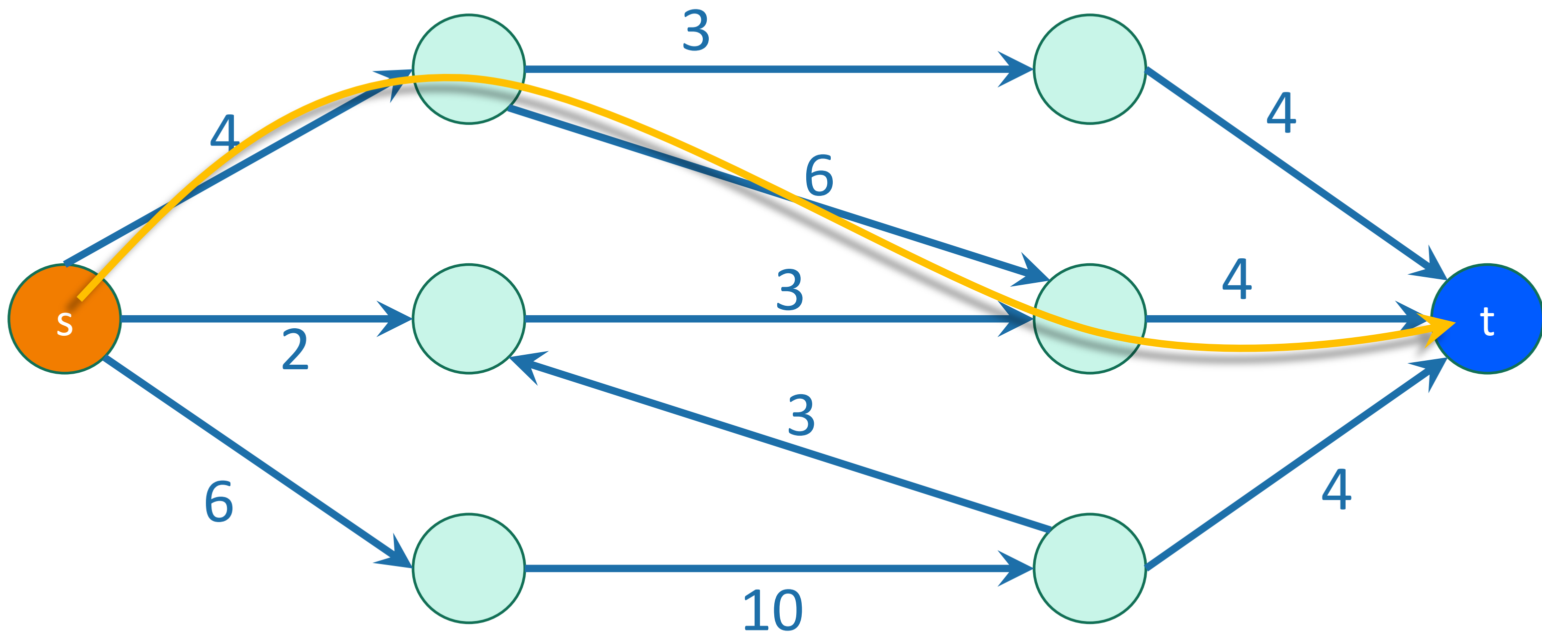
Example of Ford-Fulkerson



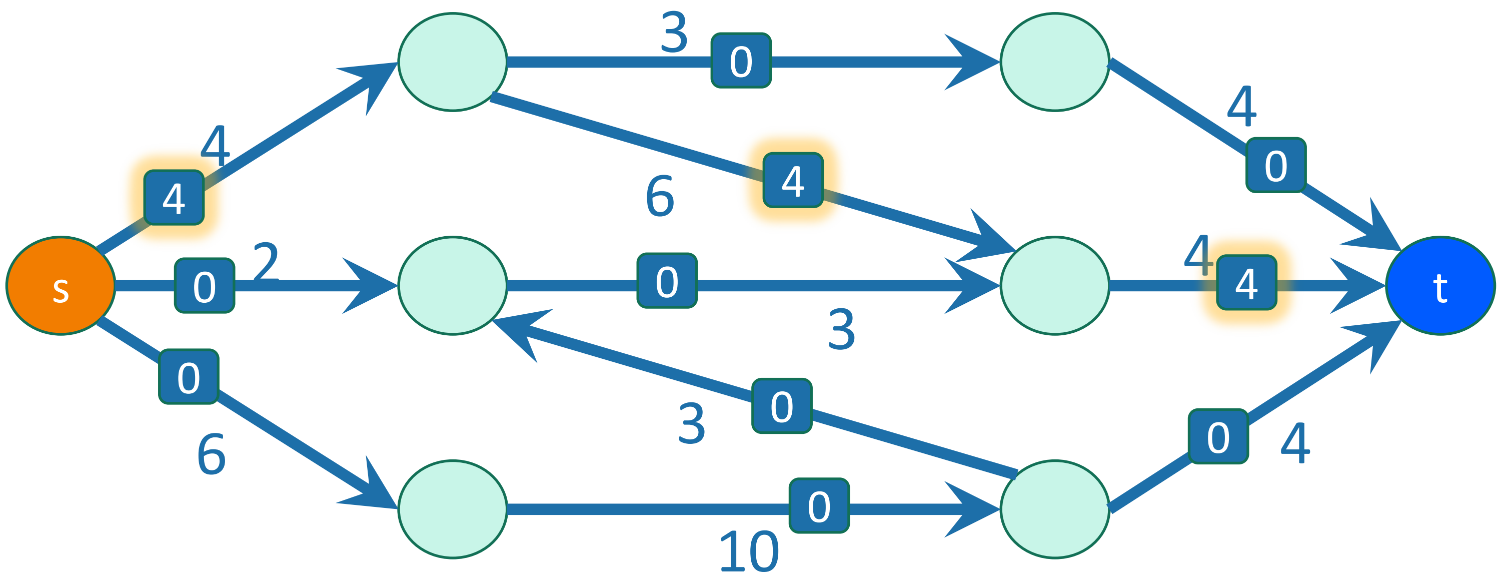


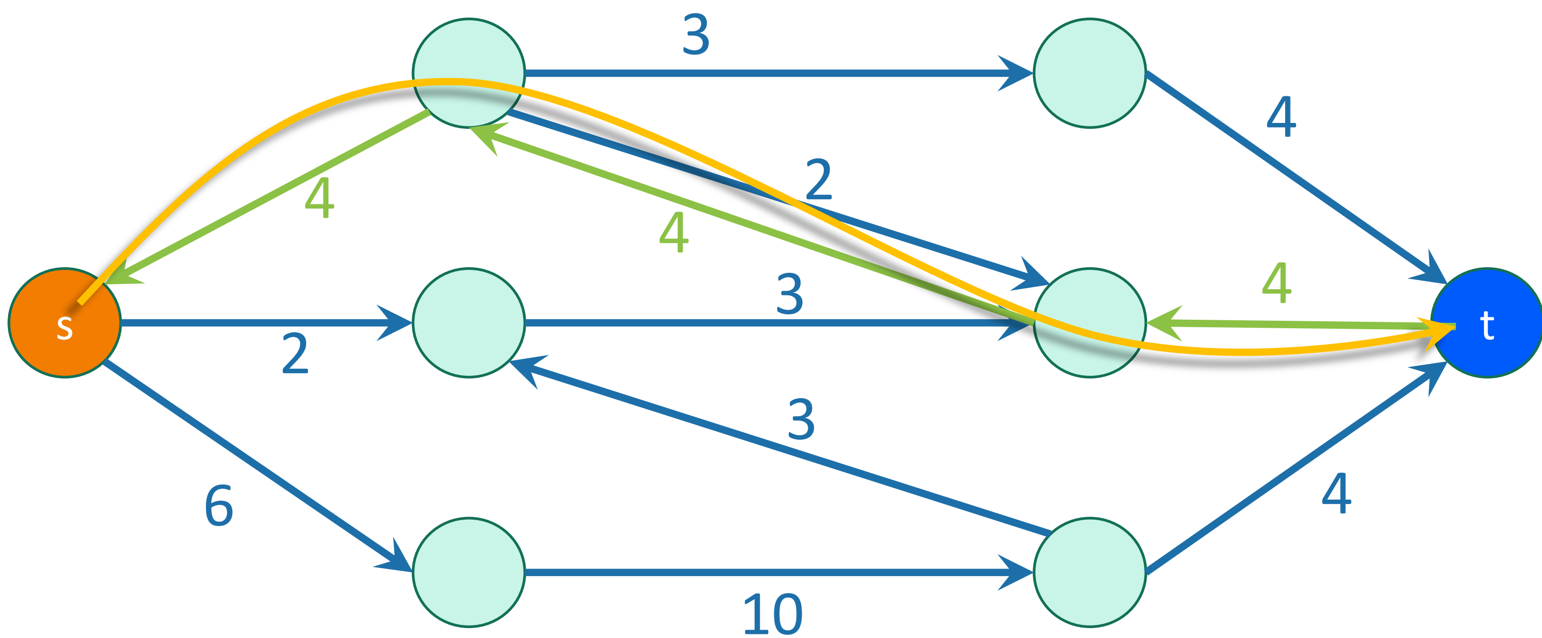
Example of Ford-Fulkerson



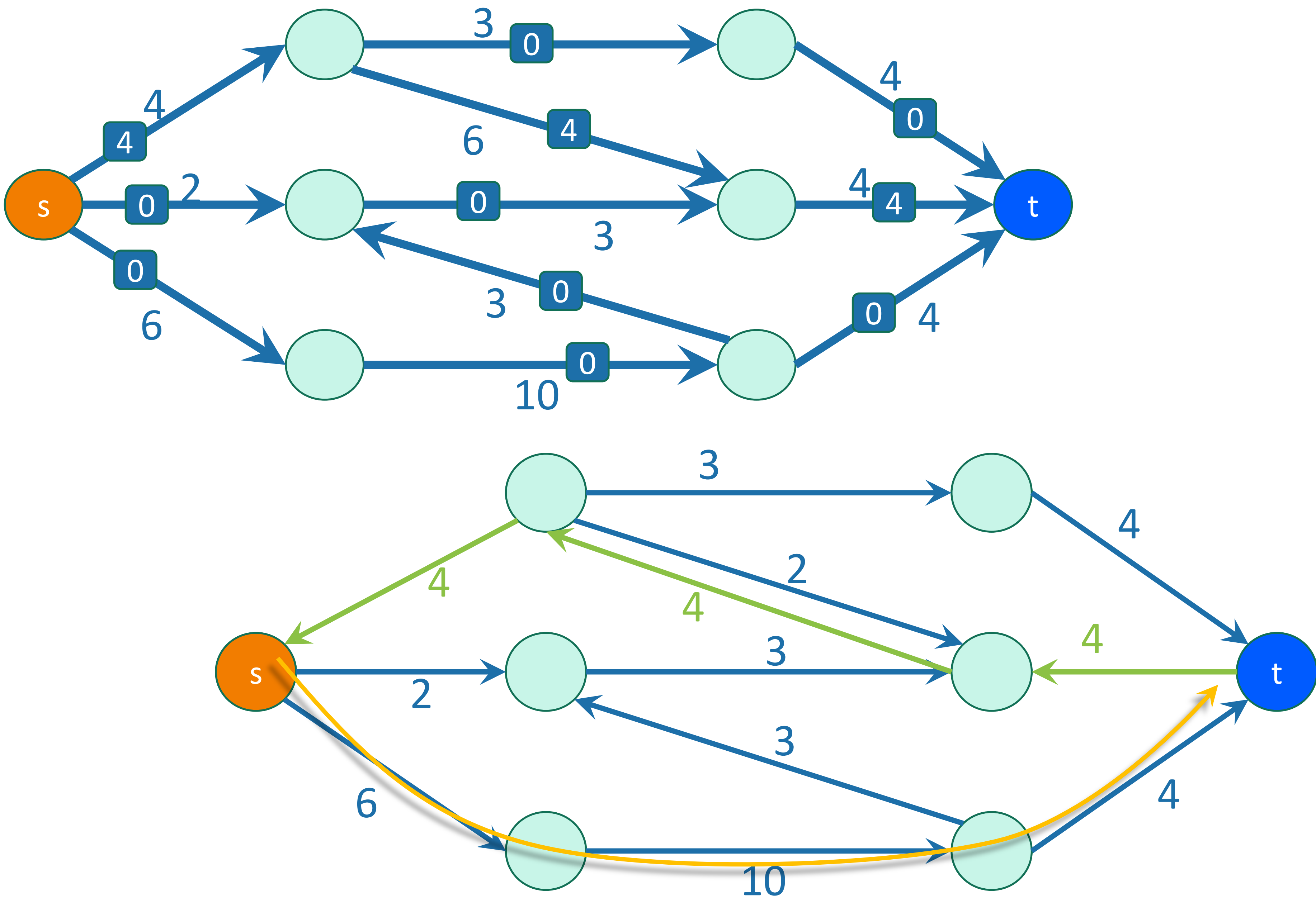


Example of Ford-Fulkerson

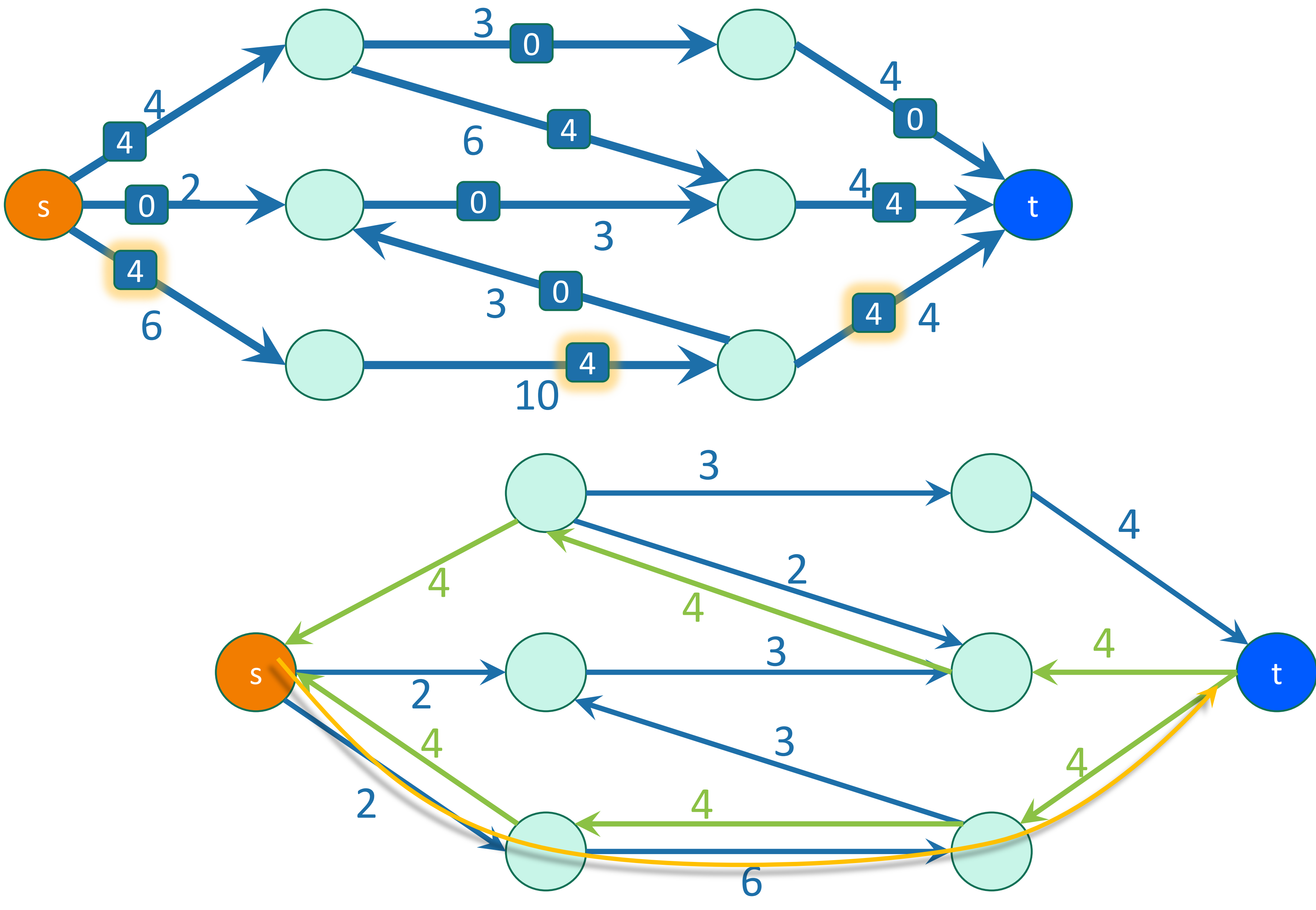




Example of Ford-Fulkerson

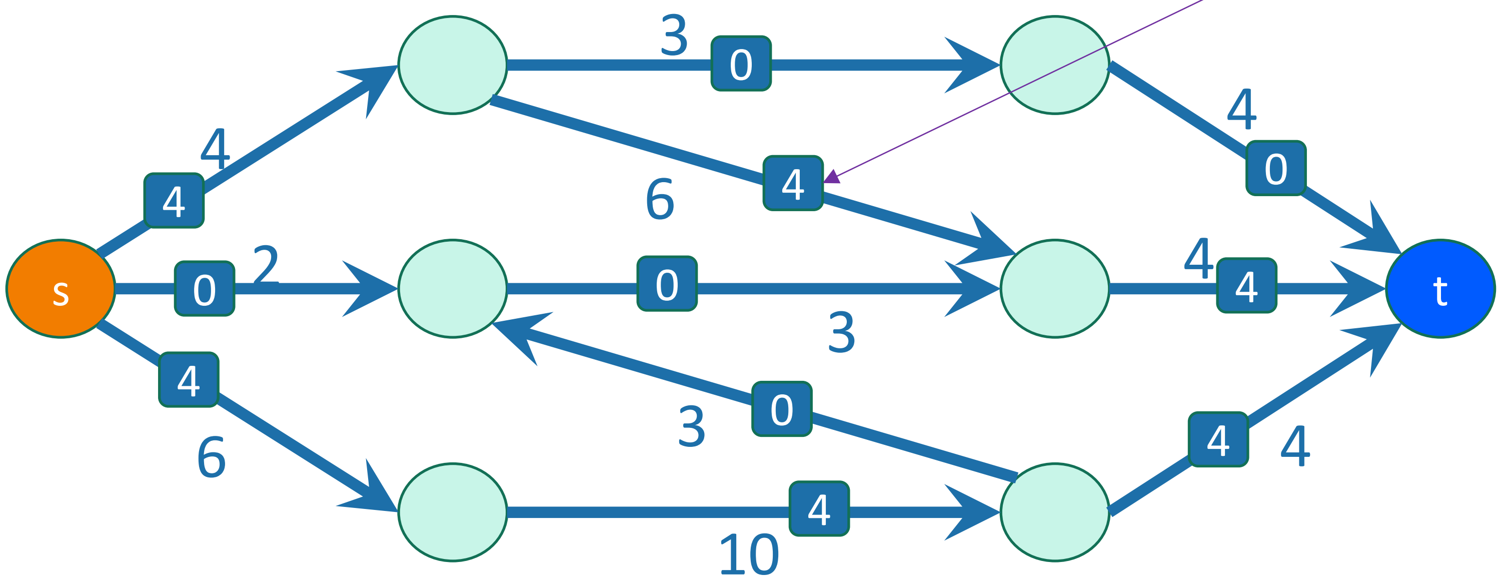


Example of Ford-Fulkerson

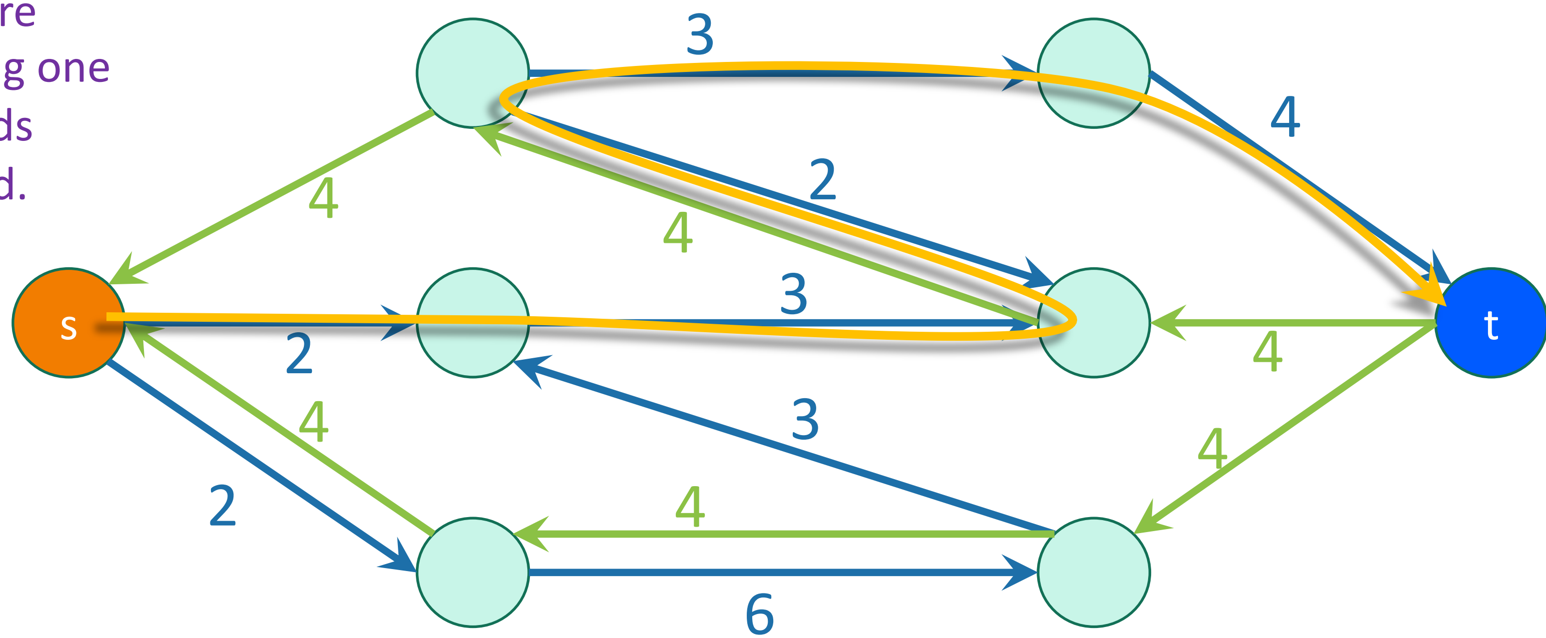


Example of Ford-Fulkerson

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

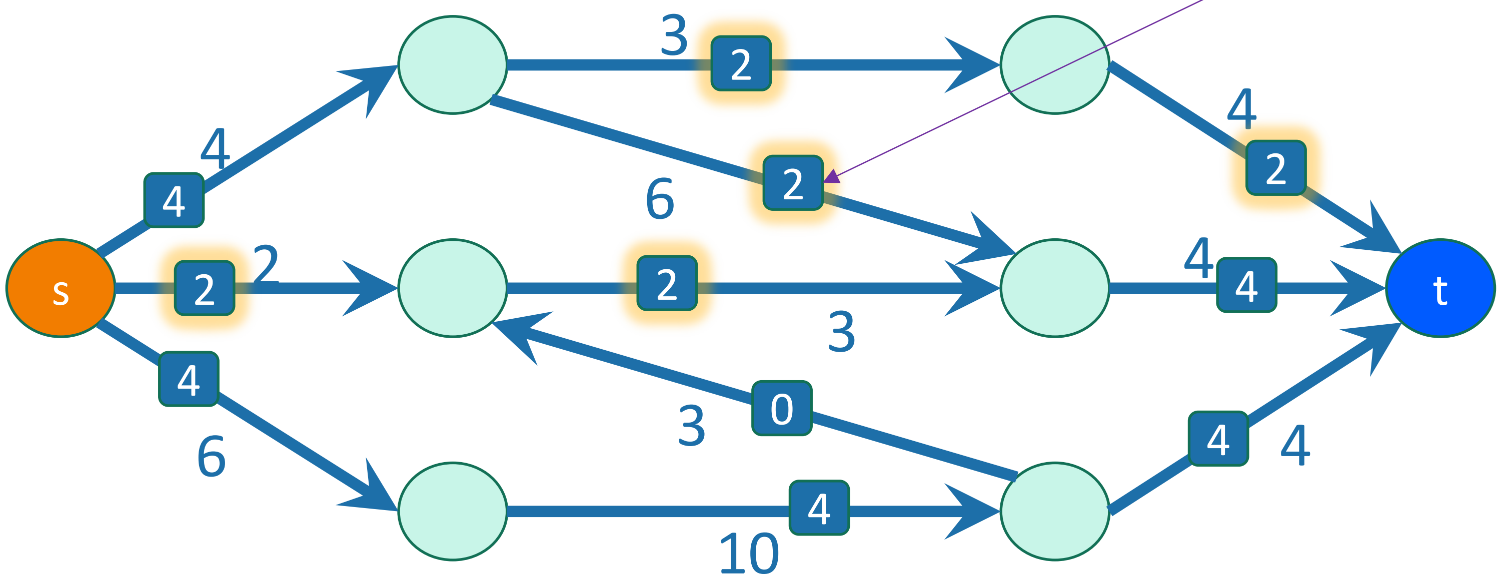


Notice that we're going back along one of the backwards edges we added.

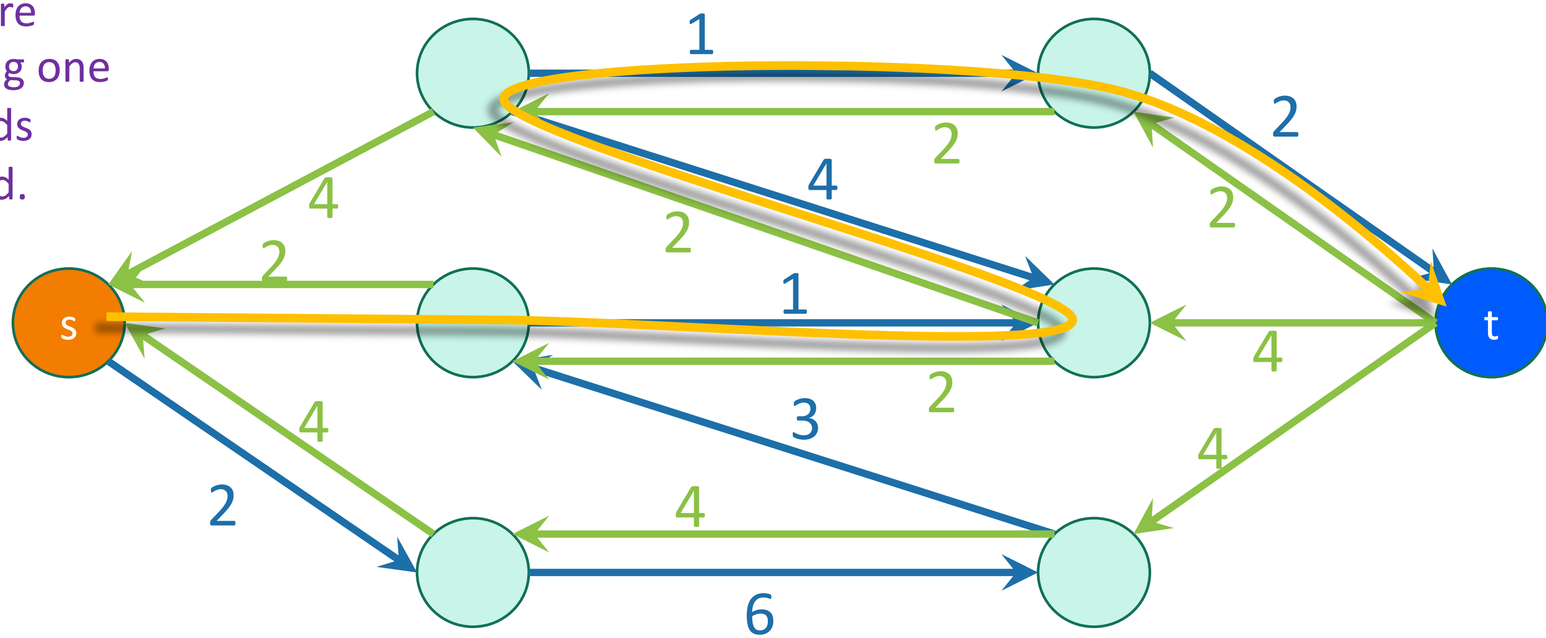


Example of Ford-Fulkerson

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

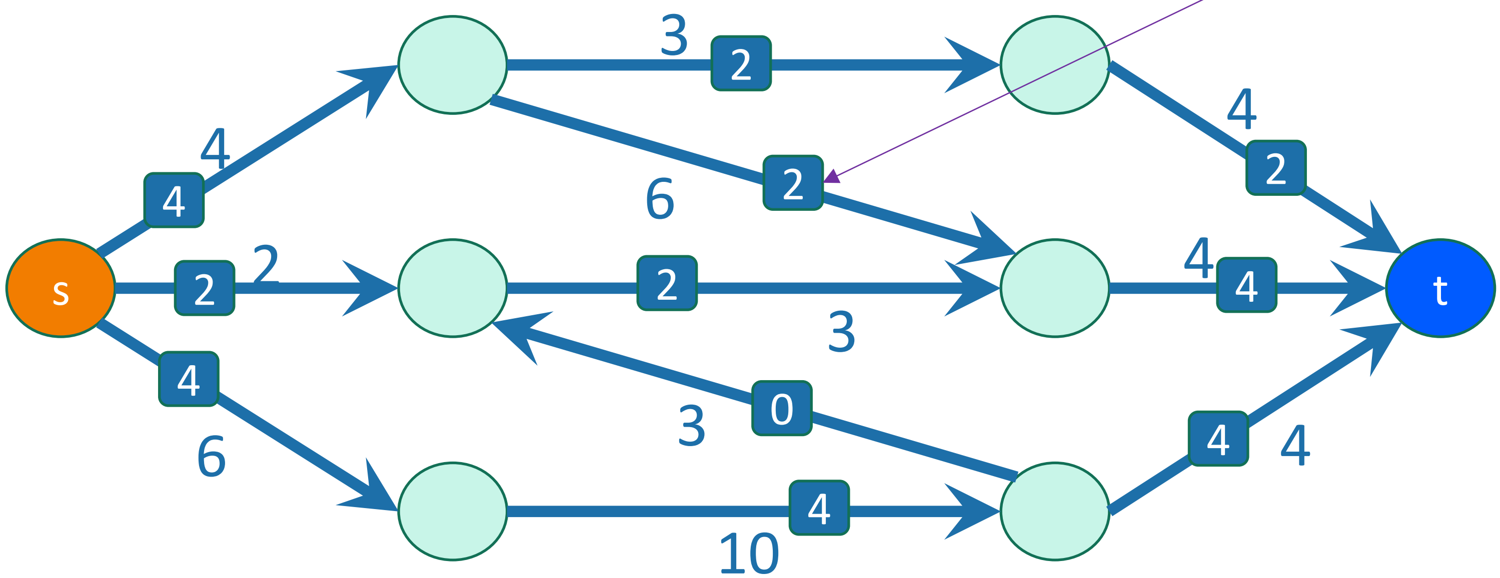


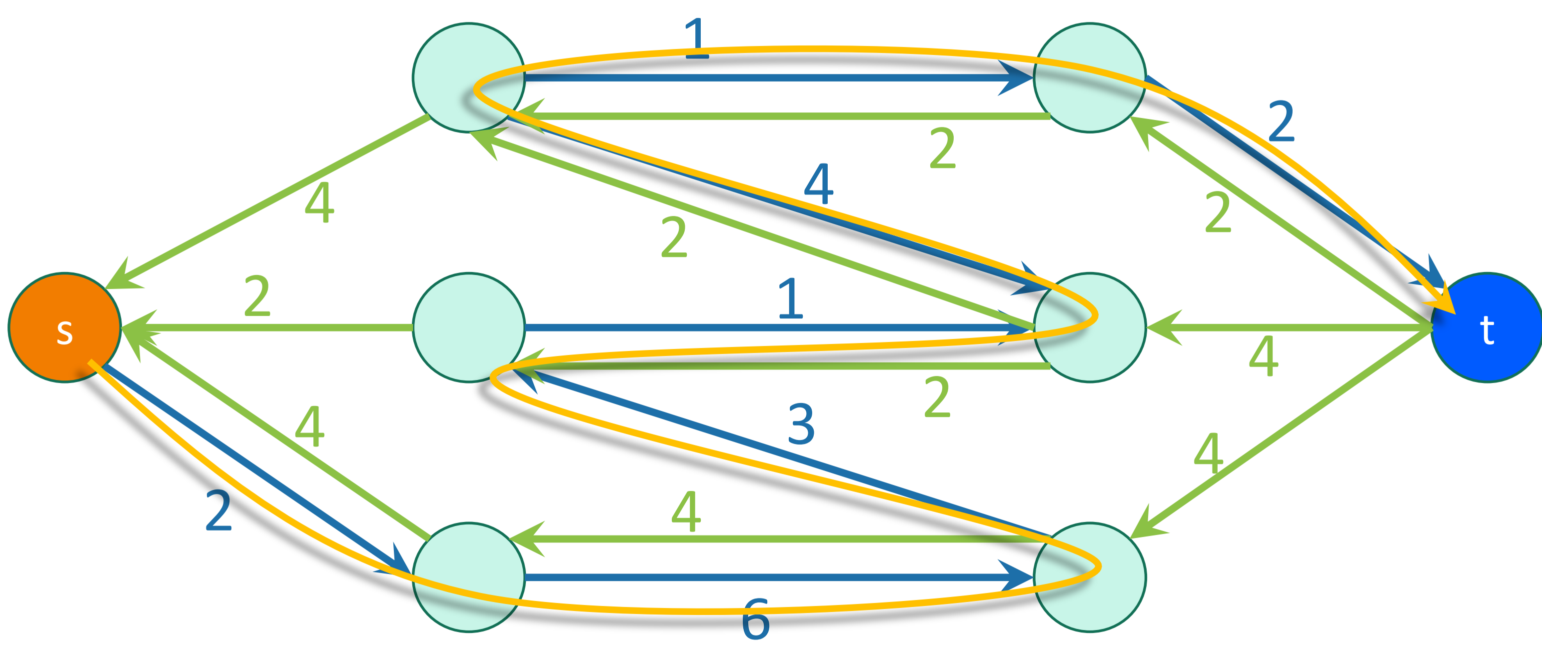
Notice that we're going back along one of the backwards edges we added.



Example of Ford-Fulkerson

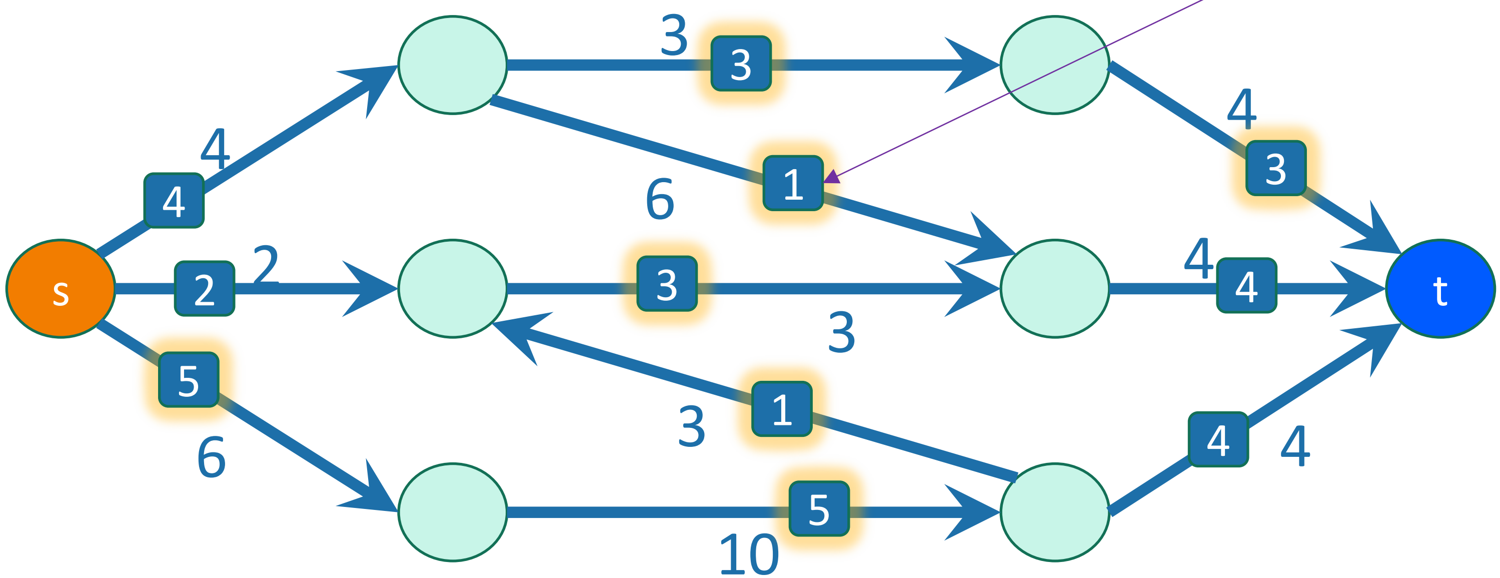
We will remove flow from this edge AGAIN.

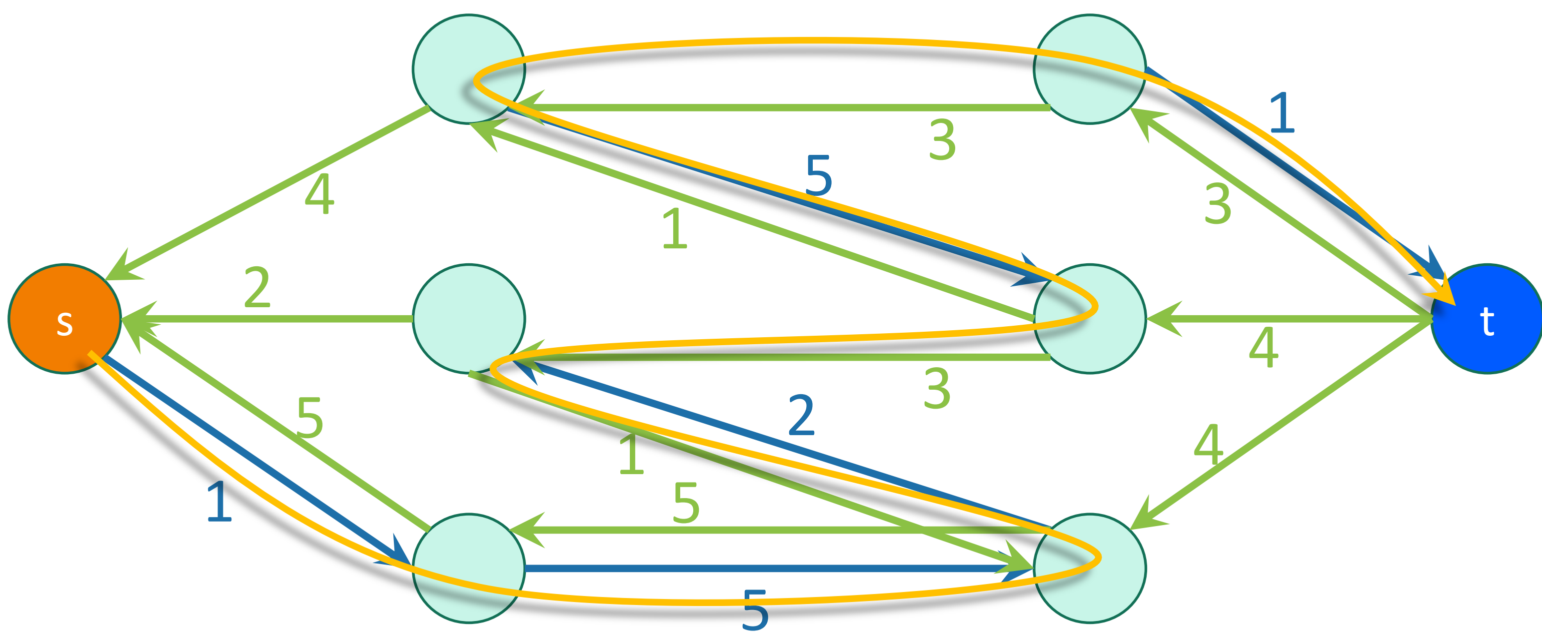




Example of Ford-Fulkerson

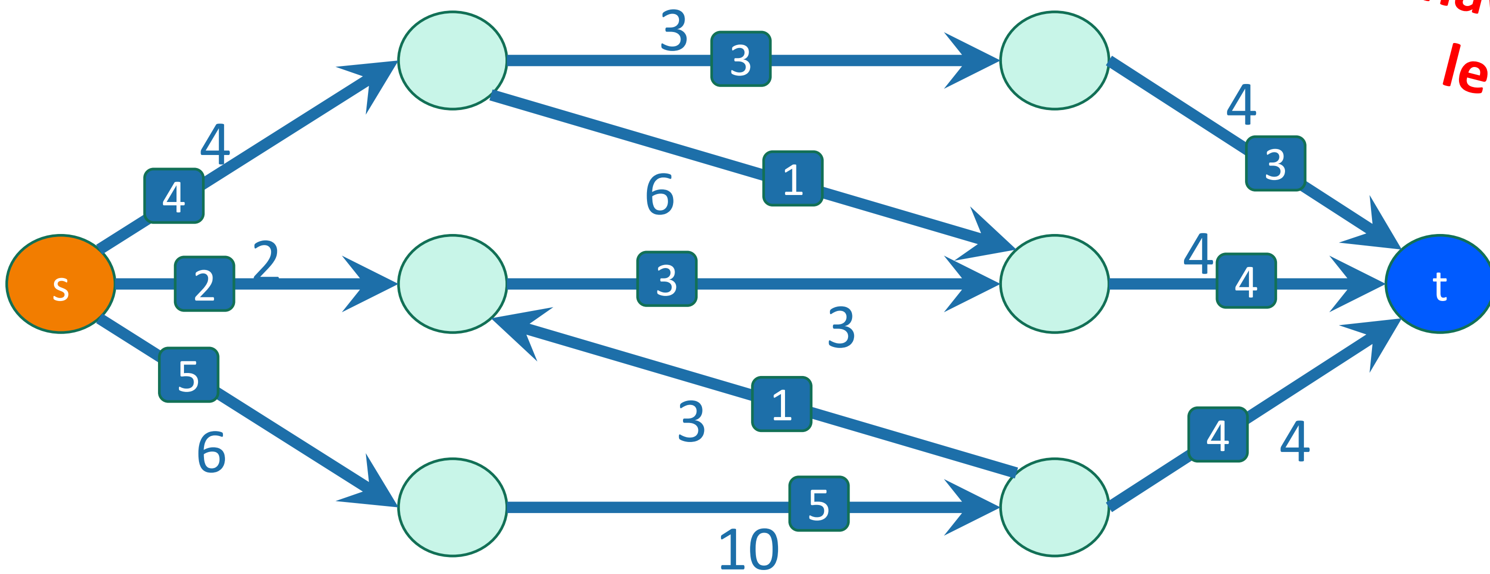
We will remove flow from this edge AGAIN.

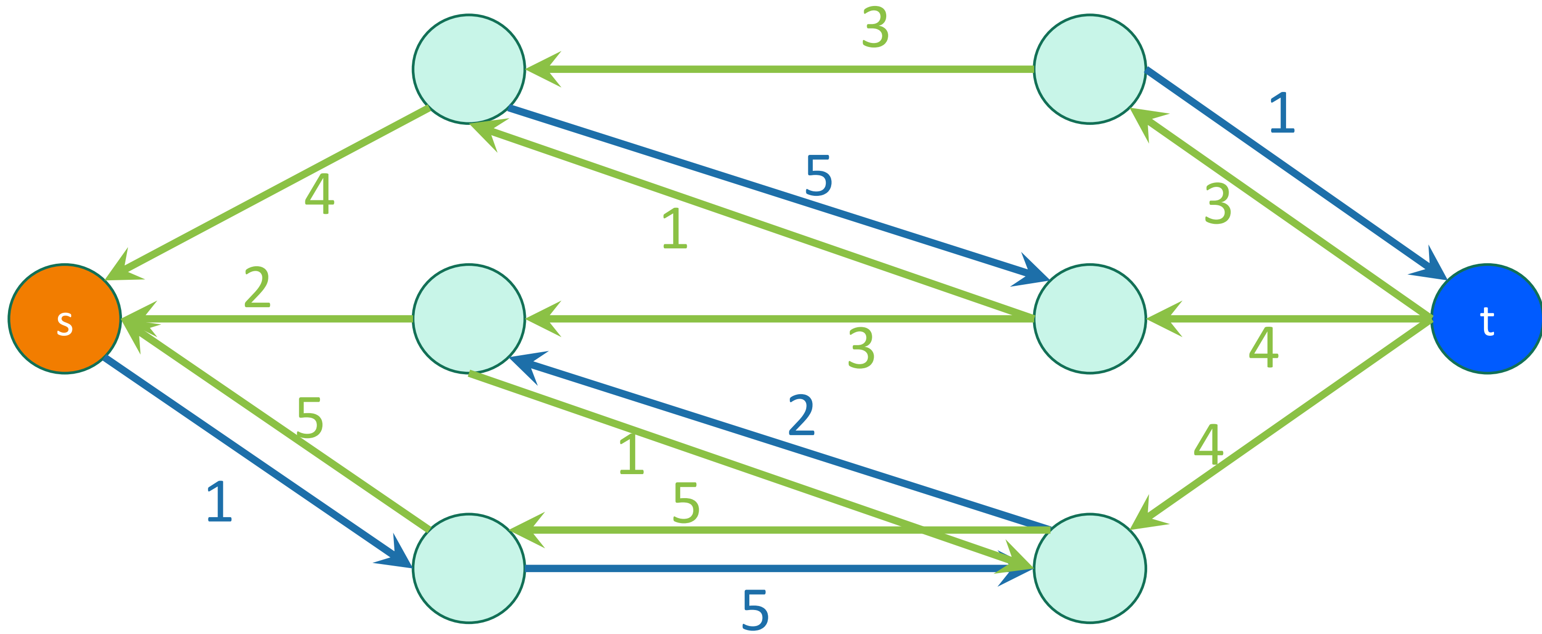




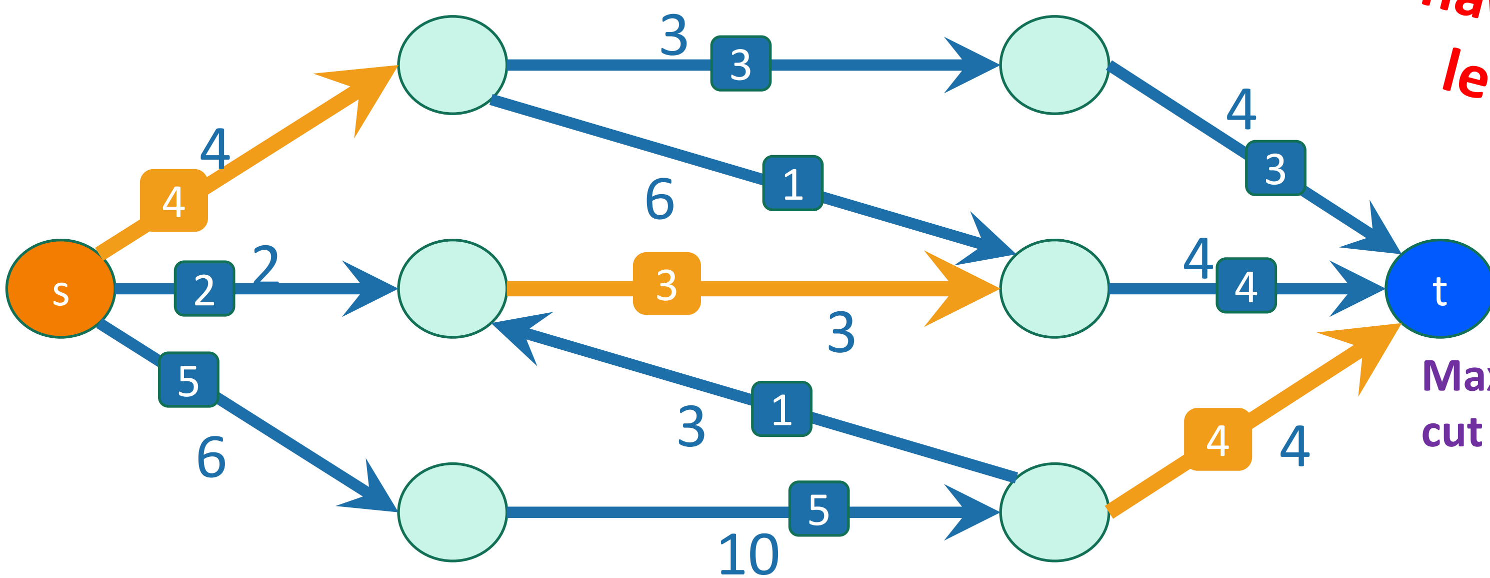
Example of Ford-Fulkerson

Now we have nothing left to do!



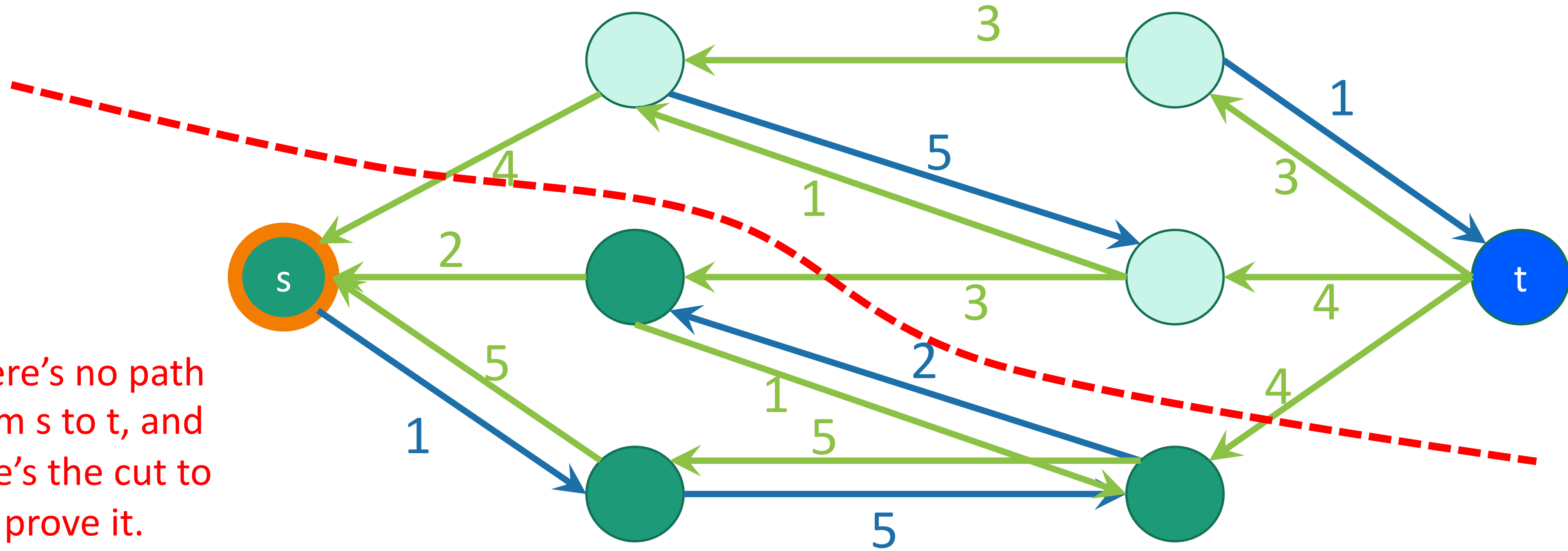


Example of Ford-Fulkerson



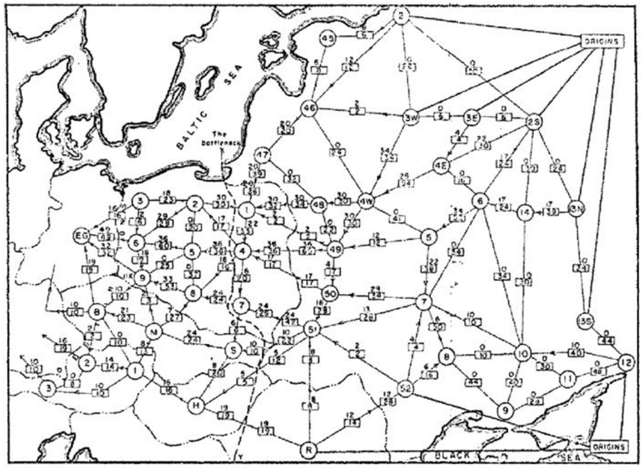
Now we have nothing left to do!

Max flow and min cut are both 11.



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…
- The Ford-Fulkerson algorithm can find the min-cut/max-flow.
  - Repeatedly improve your flow along an augmenting path.
- **How long does this take???**

Doing Ford-Fulkerson with BFS is called the **Edmonds-Karp algorithm**.

Theorem

- If you use BFS, the Ford-Fulkerson algorithm runs in time $\mathbf{O(nm^2)}$

Doesn't have anything to do with the edge weights!

- Basic idea:
  - The number of times you remove an edge from the residual graph is O(n).
    - This is the hard part
  - There are at most m edges.
  - Each time we remove an edge we run BFS, which takes time O(n+m).
    - Actually, O(m), since we don't need to explore the whole graph, just the stuff reachable from s.

One more useful thing

- 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 integral.

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!