CSE100: Design and Analysis of Algorithms
Minimum Spanning Trees and Max Flow

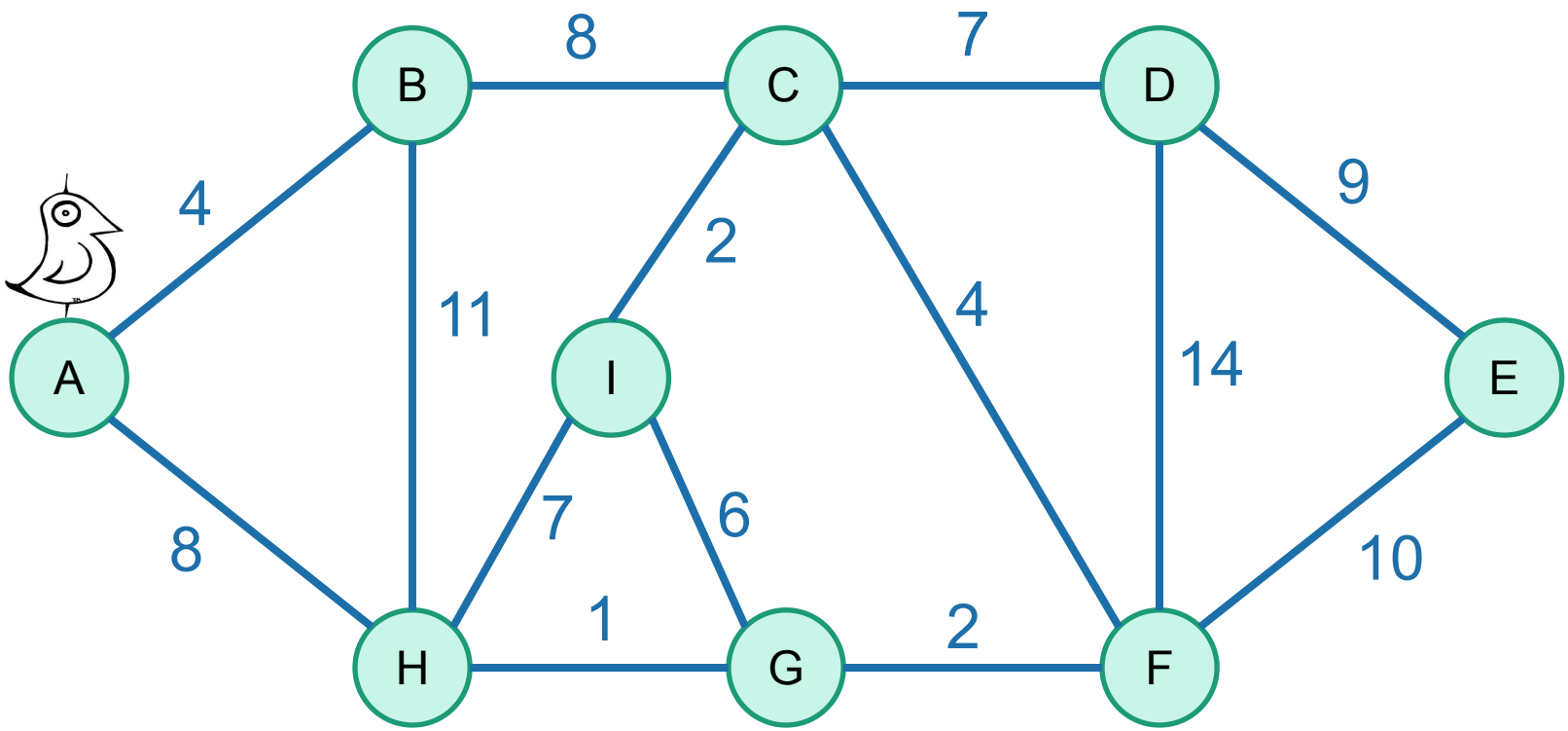
Kruskal, Max Flows, Min Cuts, and Ford-Fulkerson Algorithms

What have we learned?

- Prim's algorithm greedily grows a tree
  - smells a lot like Dijkstra's algorithm

Prim's algl:

Start growing a tree, greedily add the shortest edge we can to grow the tree.

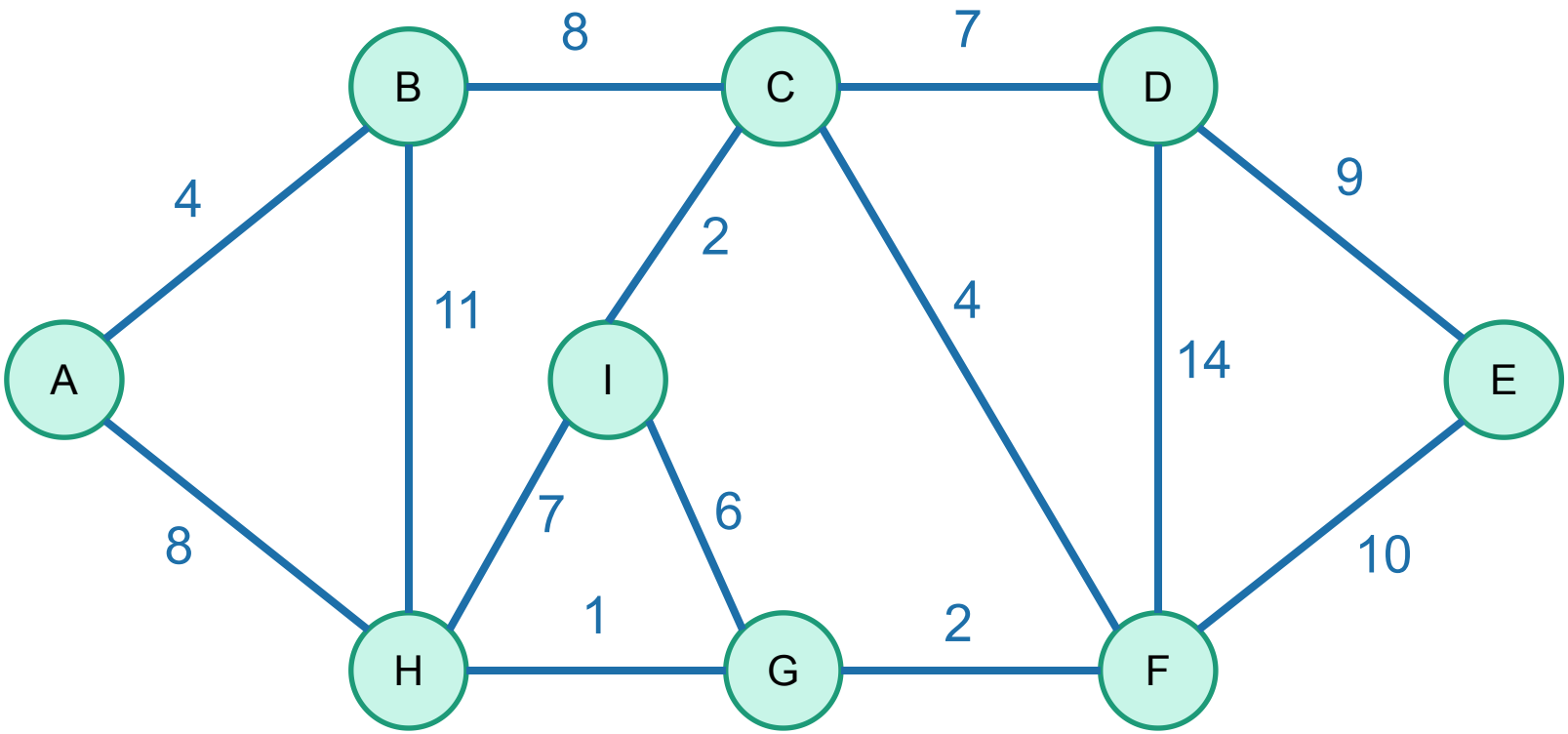


That’s not the only greedy algorithm for MST!

That's not the only greedy algorithm

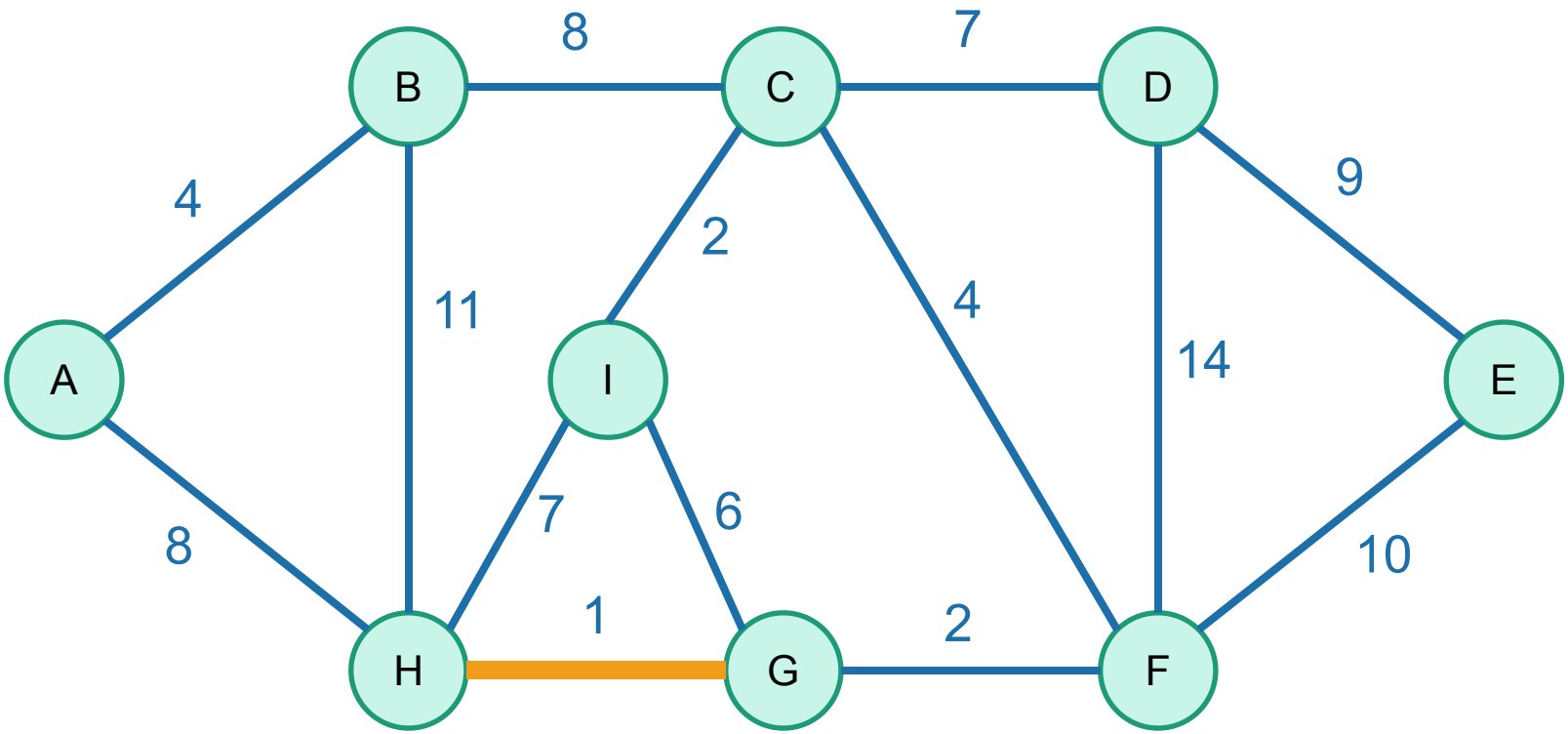
what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?



That's not the only greedy algorithm

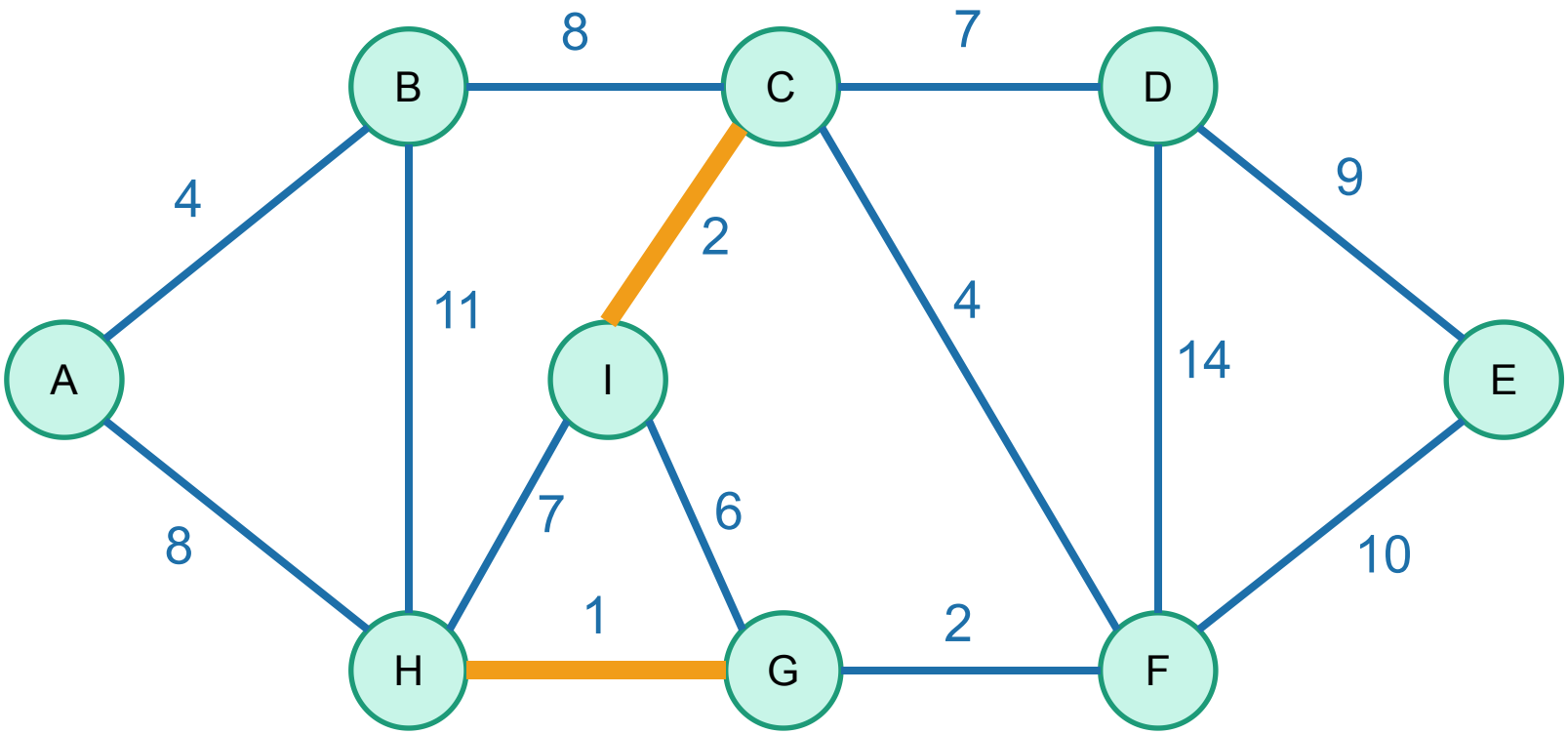
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

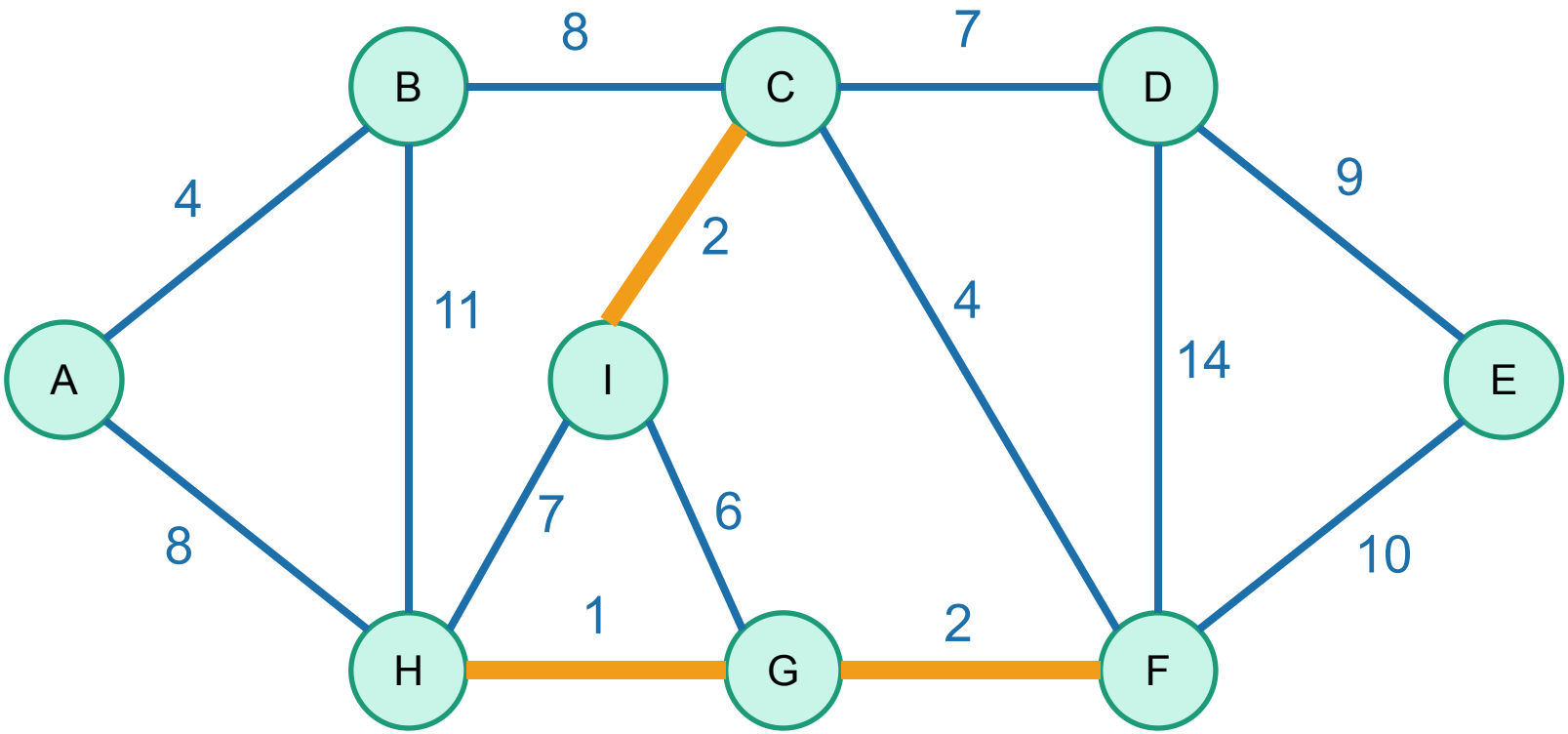
what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?



That's not the only greedy algorithm

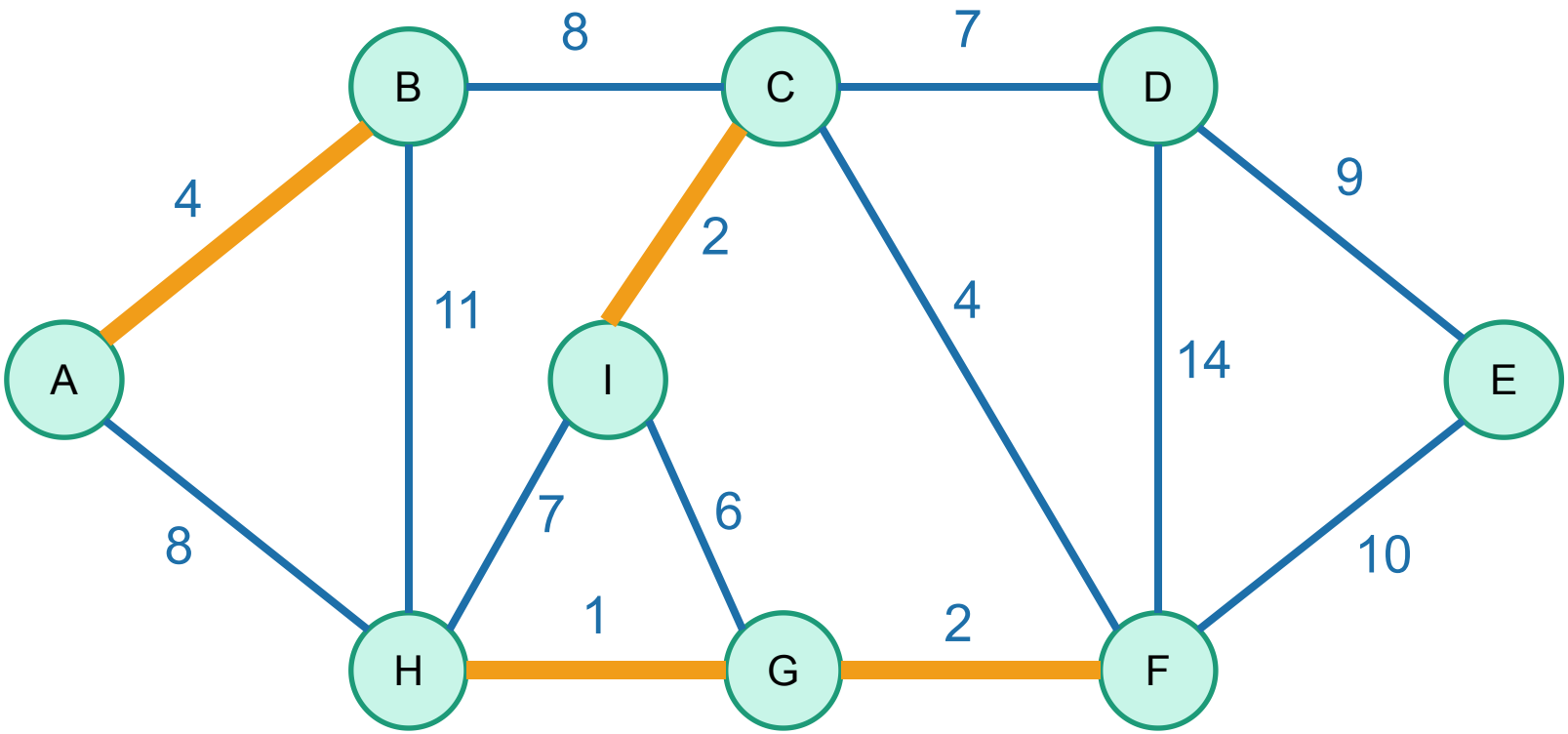
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

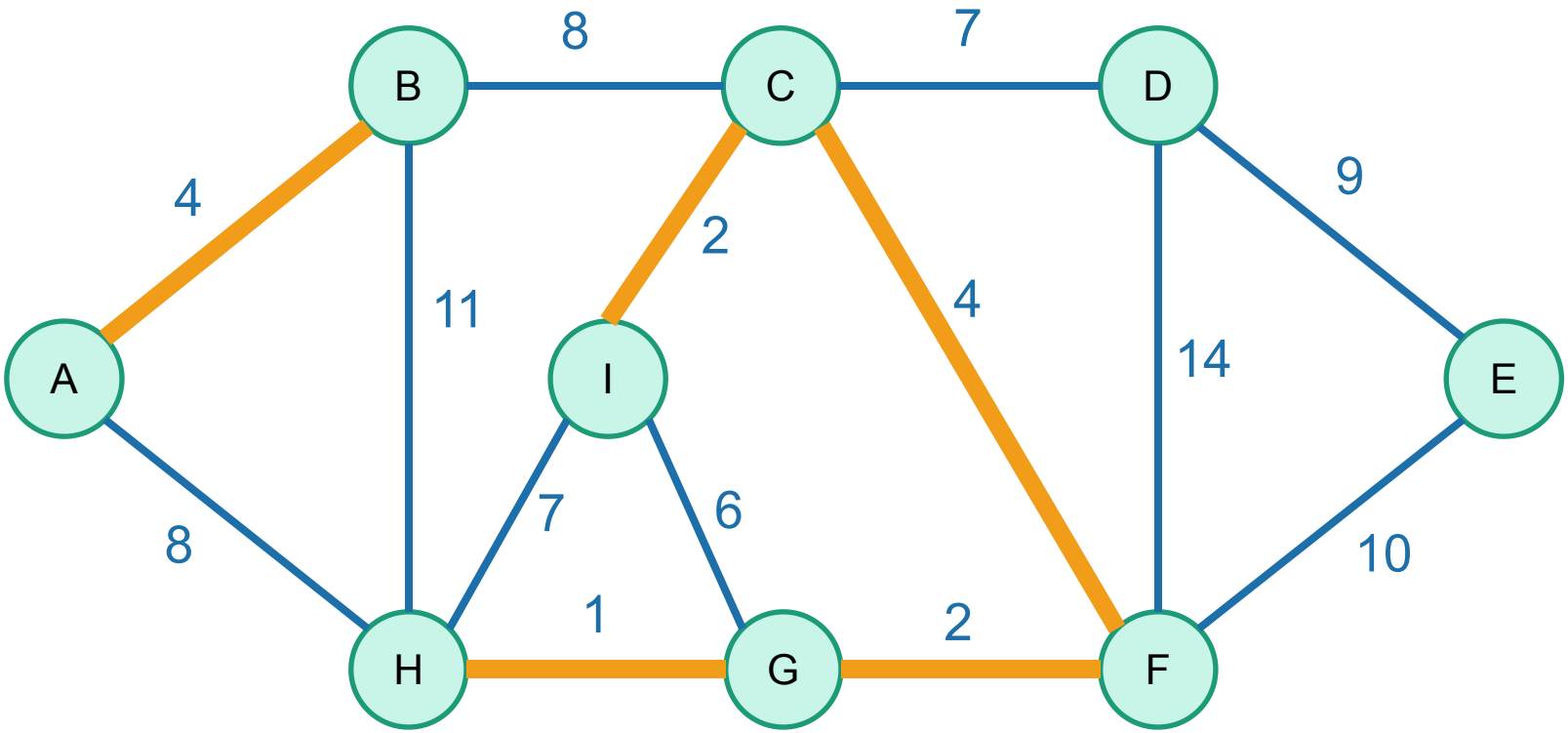
what if we just always take the cheapest edge?

whether or not it's connected to what we have so far?



That's not the only greedy algorithm

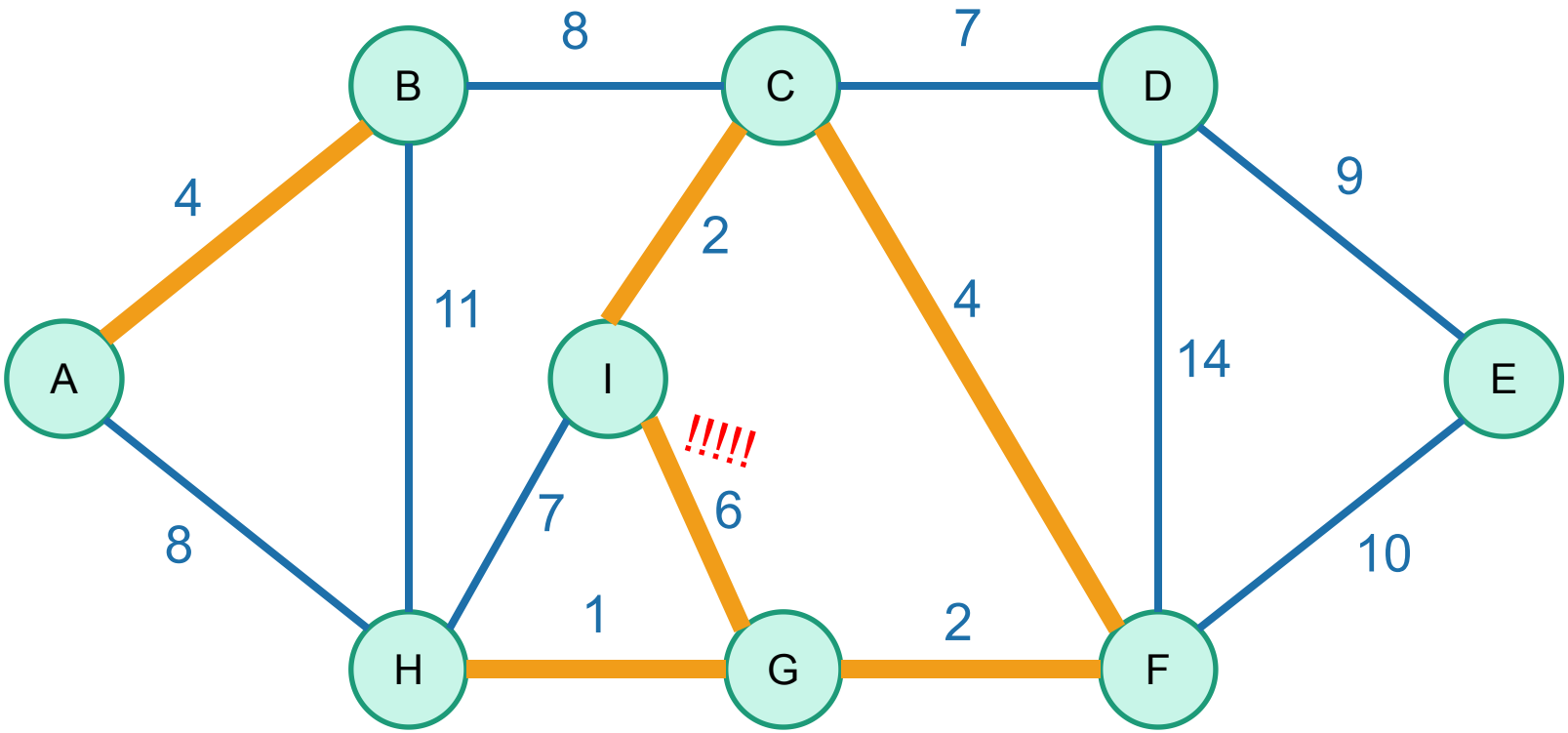
what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

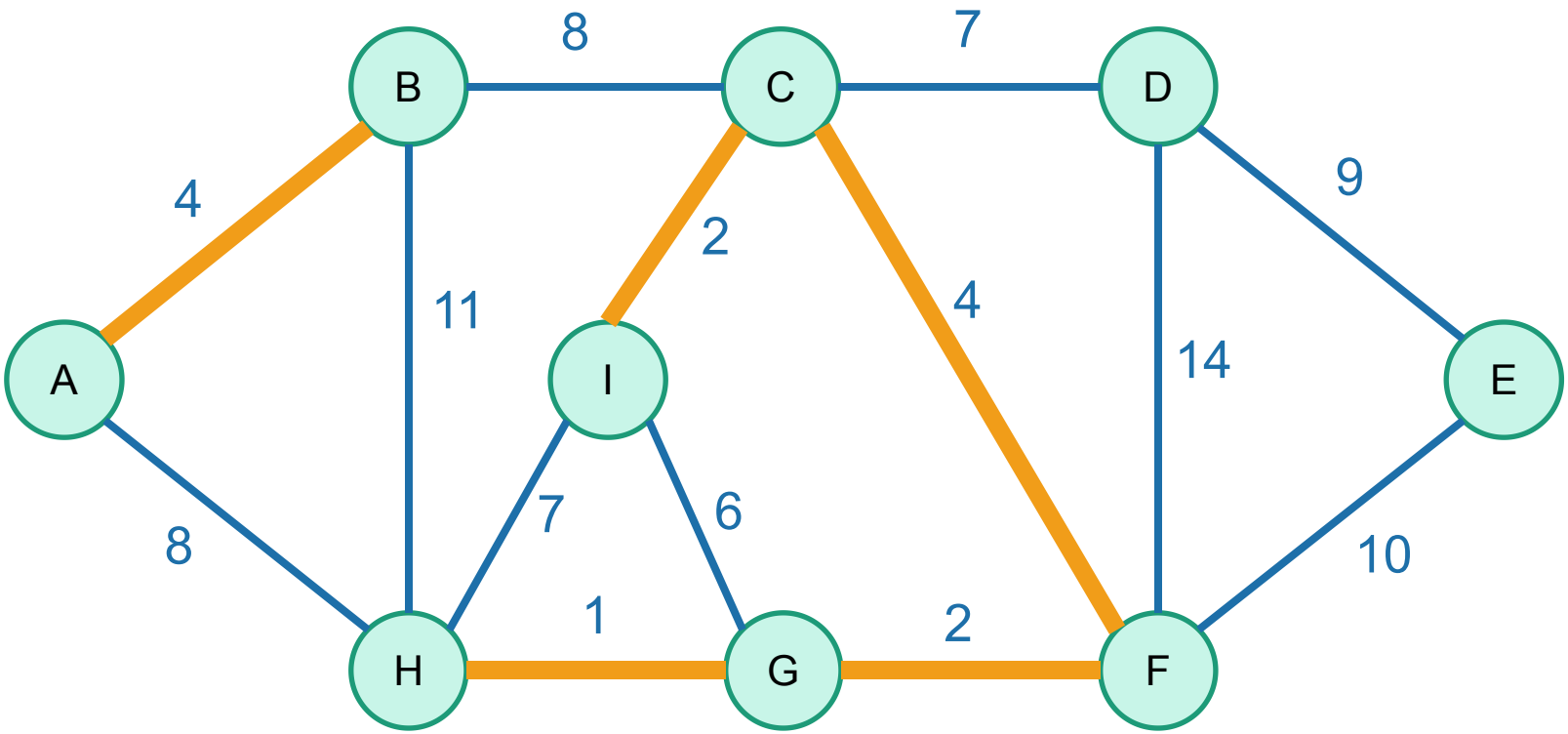
Such that it won't cause a cycle



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

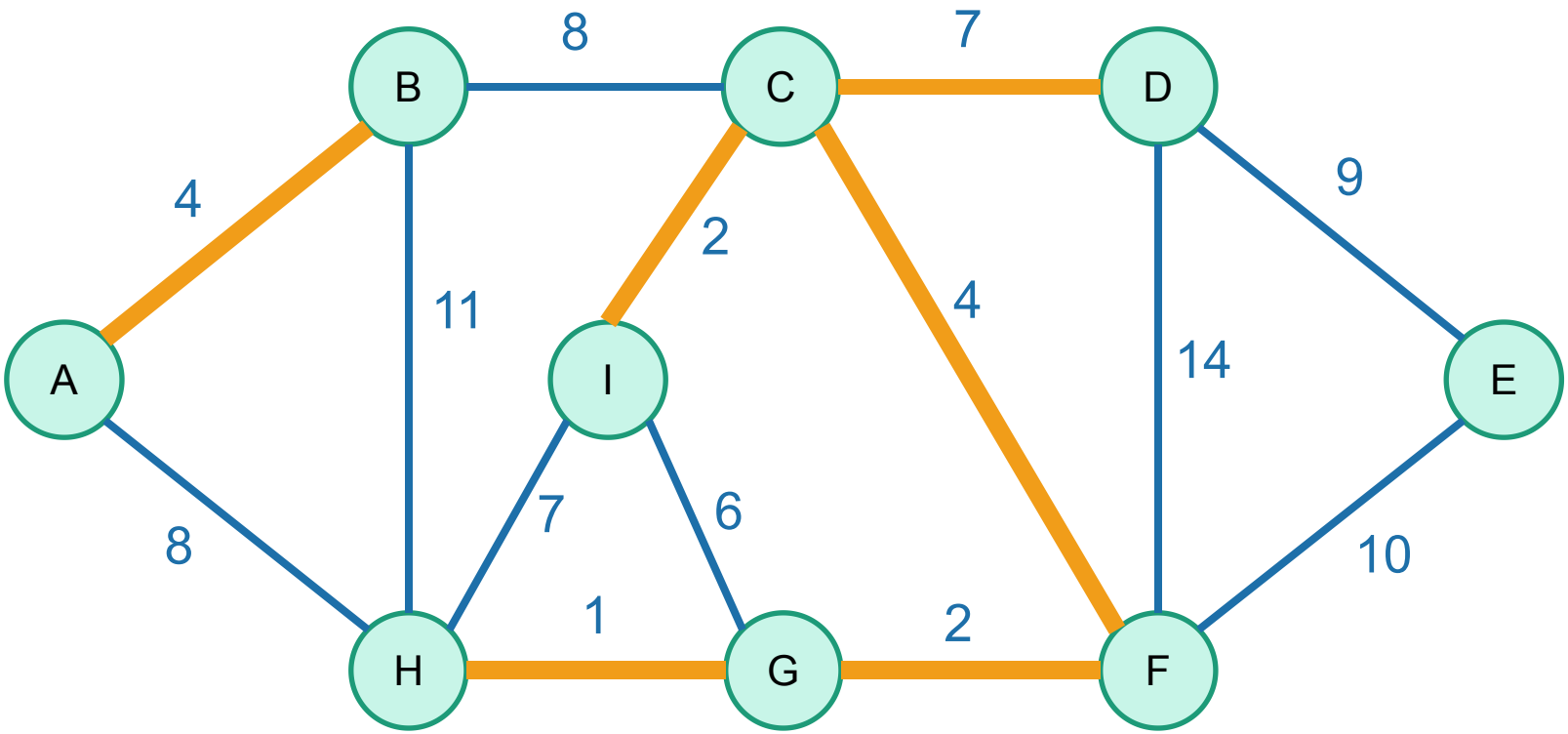
That won't caus
a cycle



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

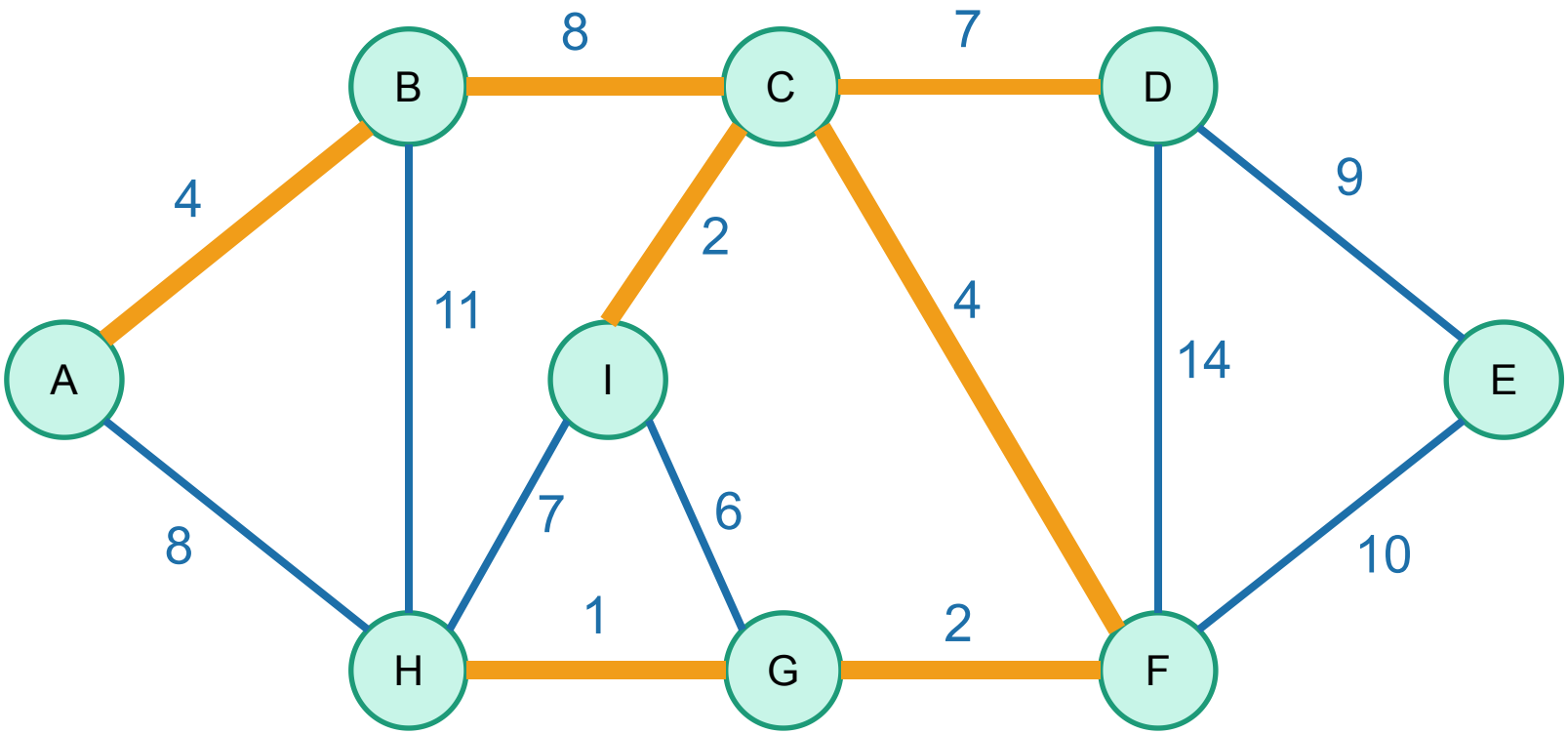
That won't caus
a cycle



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

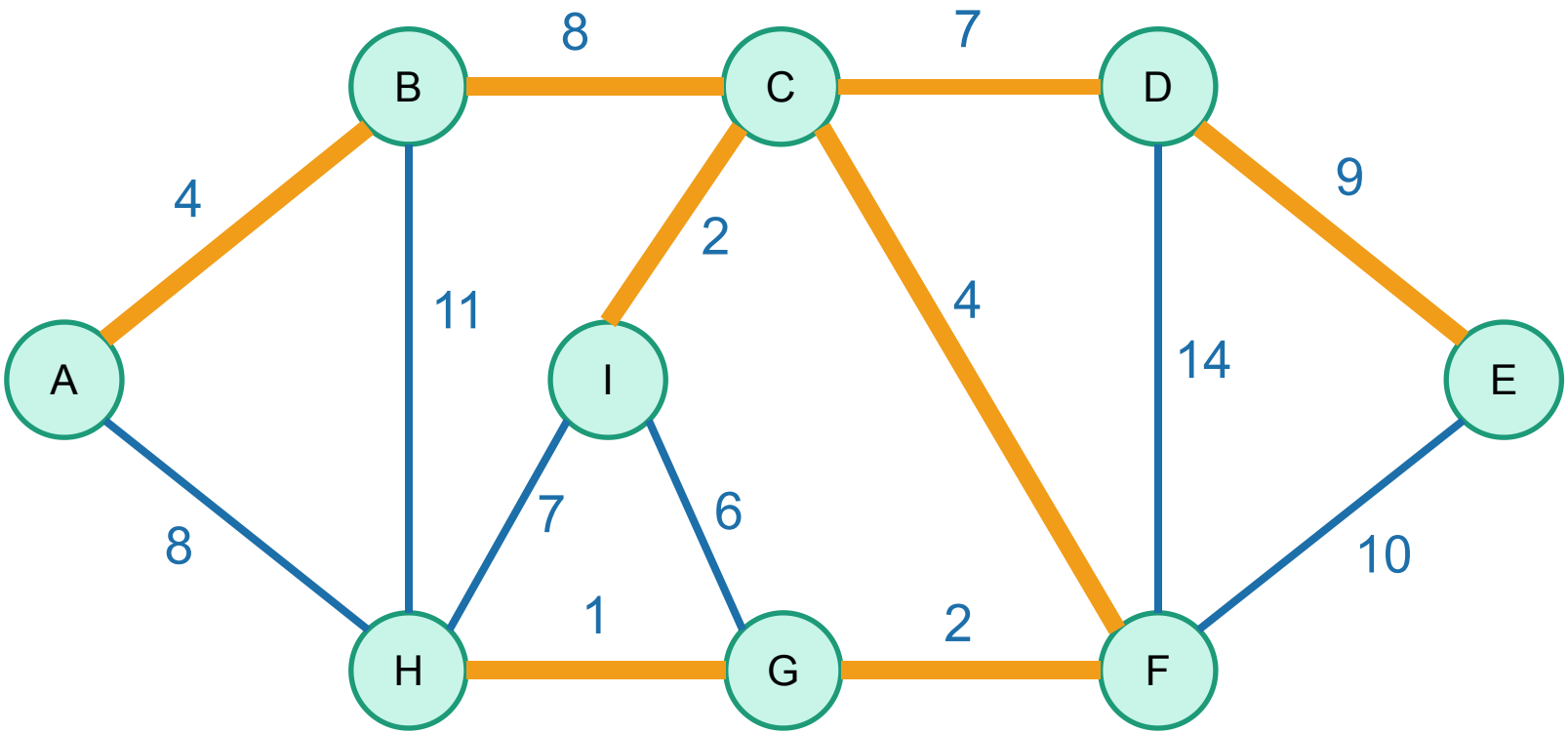
That won't caus
a cycle



That's not the only greedy algorithm

what if we just always take the cheapest edge?
whether or not it's connected to what we have so far?

That won't caus
a cycle



We've discovered

Kruskal's algorithm!

- **slowKruskal**(G = (V,E)):
  - Sort the edges in E by non-decreasing weight.
  - MST = {}
  - **for** e in E (in sorted order):
    - **if** adding e to MST won't cause a cycle:
      - add e to MST.
  - **return** MST

m iterations through this loop

How do we check this?

How **would** you figure out if added e would make a cycle in this algorithm?

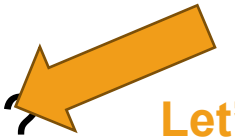
Naively, the running time is ???:

- For each of m iterations of the for loop:
  - Check if adding e would cause a cycle…

Two questions

1. Does it work?
   - That is, does it actually return a MST?

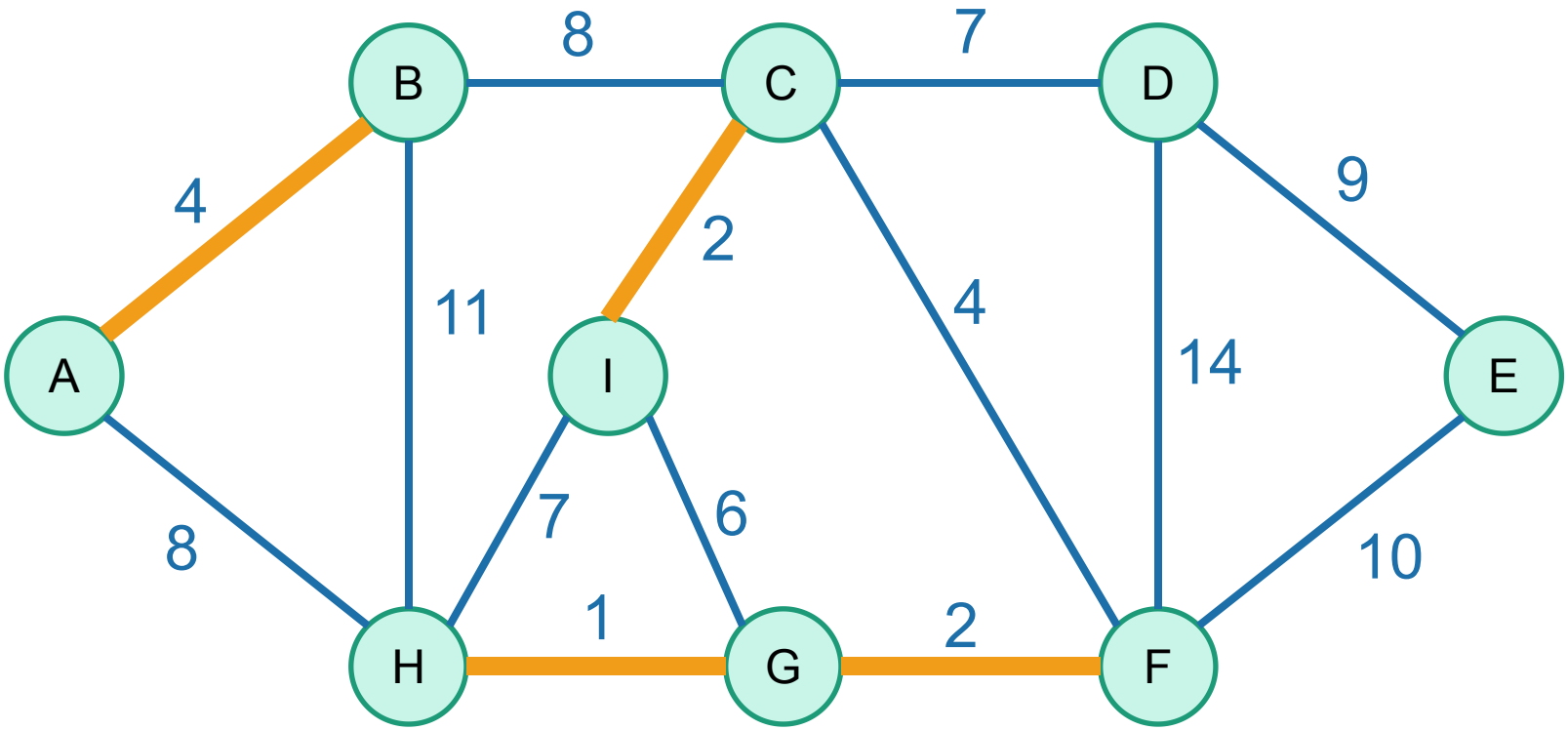
2. How do we actually implement this?
   - the pseudocode above says “slowKruskal”…



Let’s do this one first

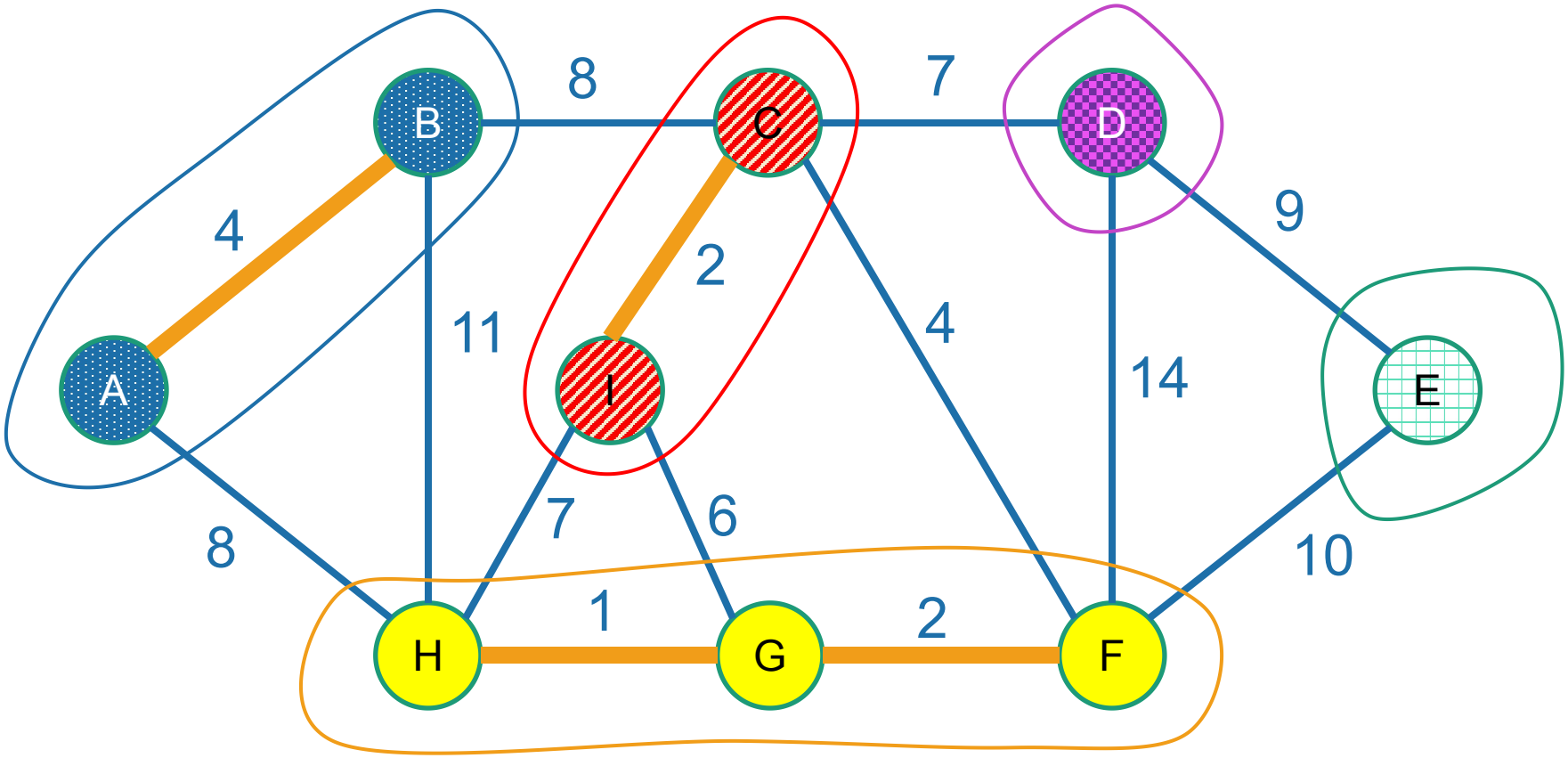
At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

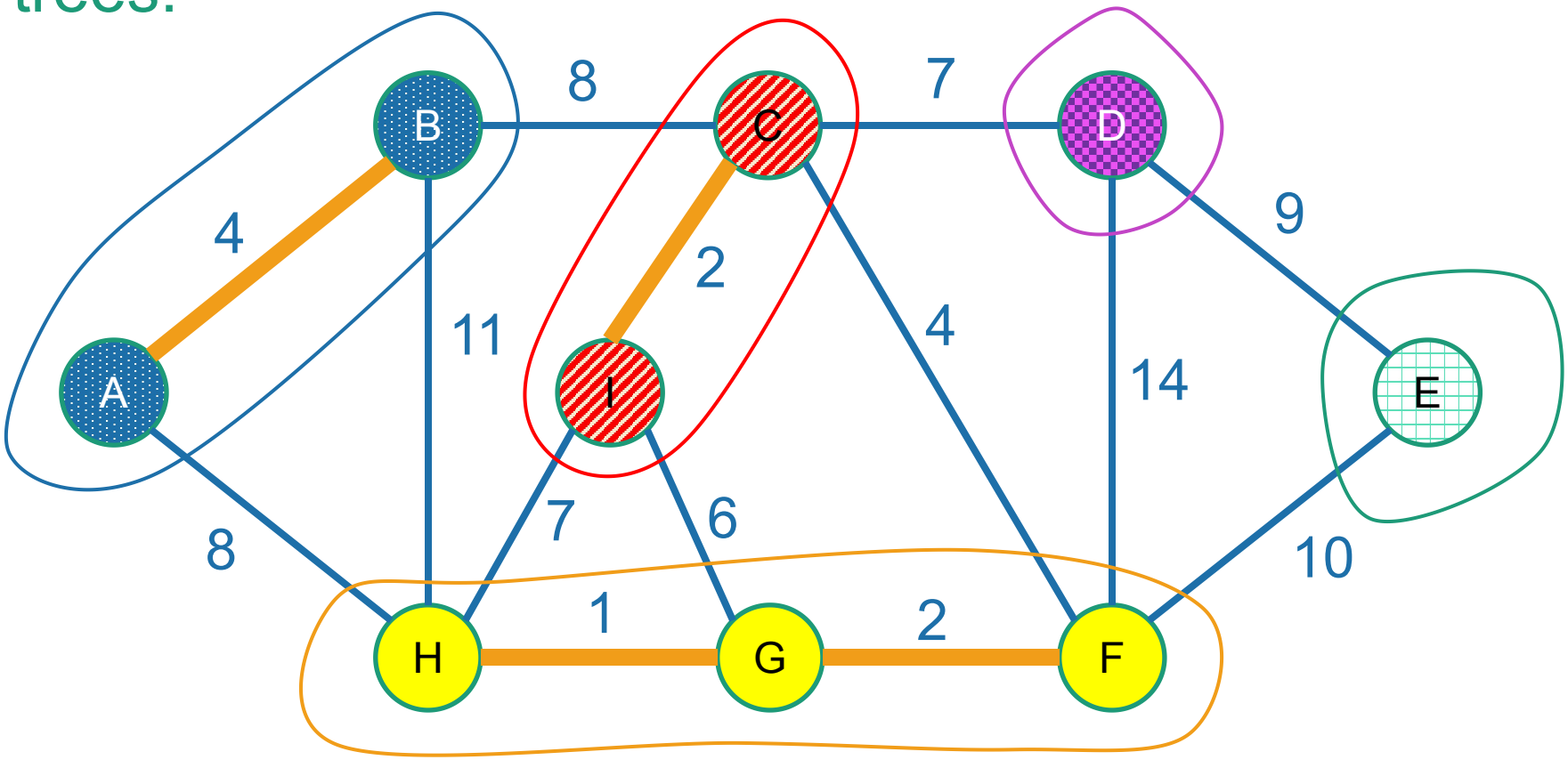
A **forest** is a collection of disjoint trees



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

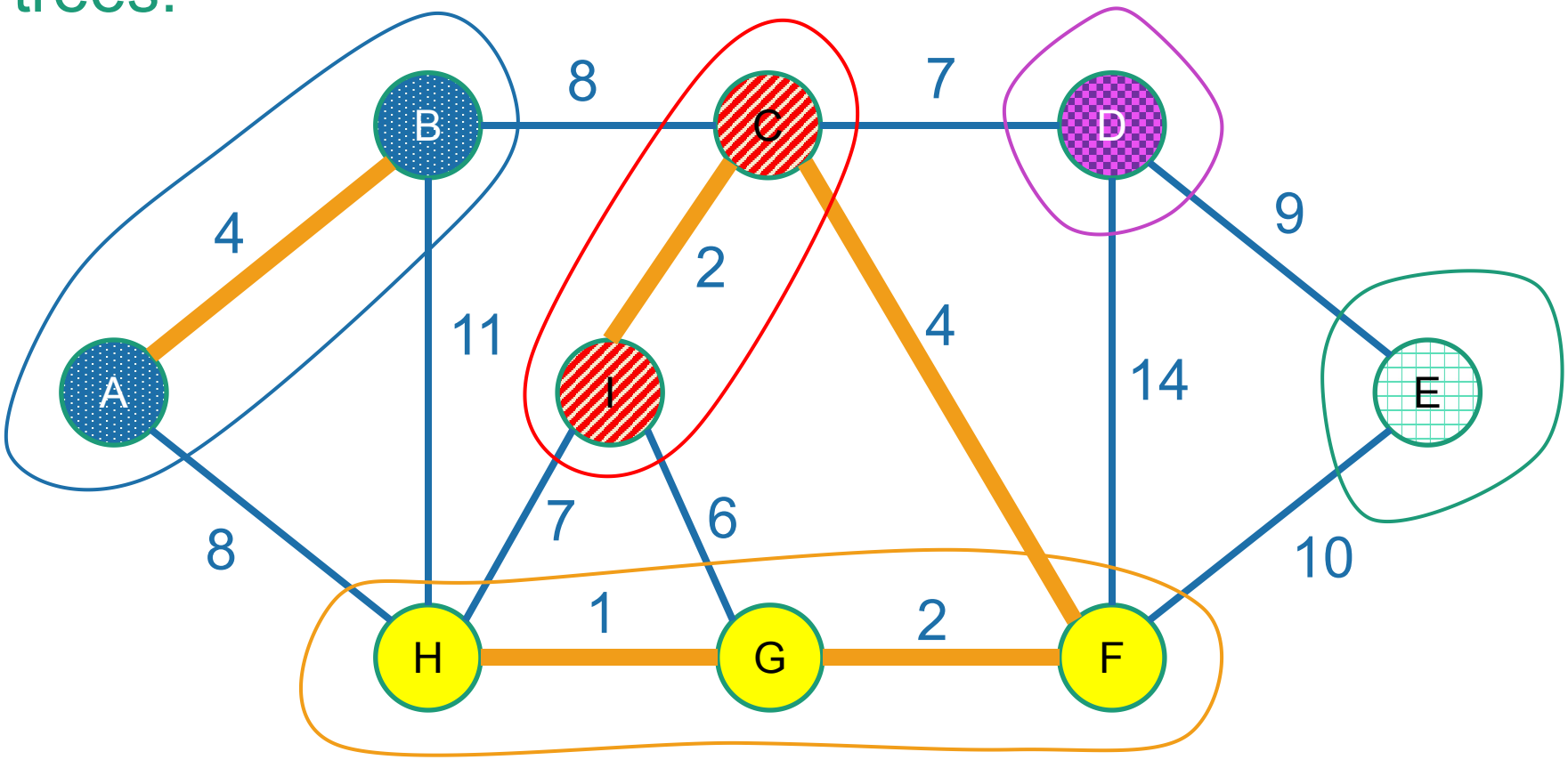
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

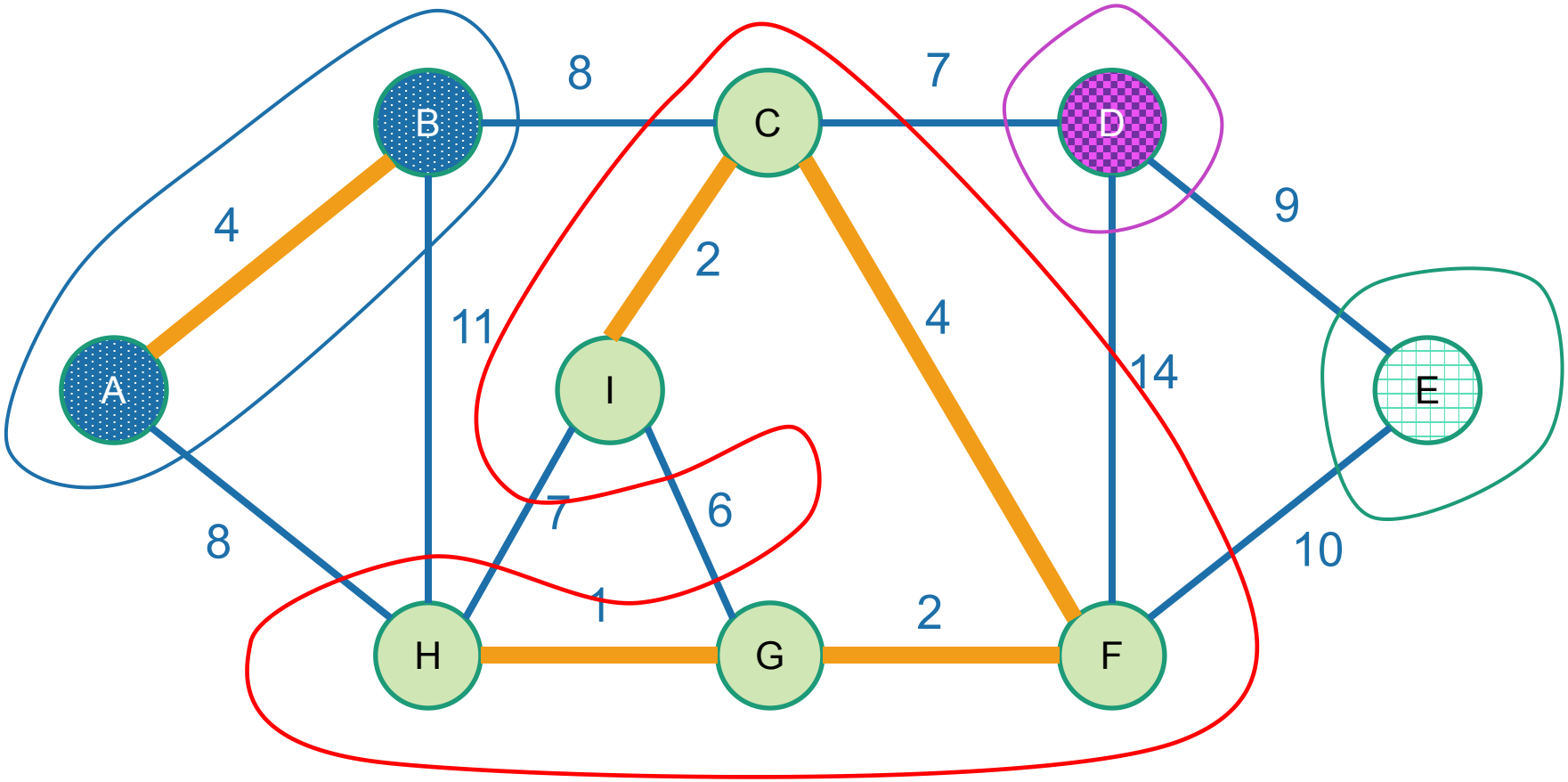
When we add an edge, we merge two trees:



At each step of Kruskal's, we are maintaining a forest.

A **forest** is a collection of disjoint trees

When we add an edge, we merge two trees:



We never add an edge within a tree since that would create a cycle.

Keep the trees in a special data structure

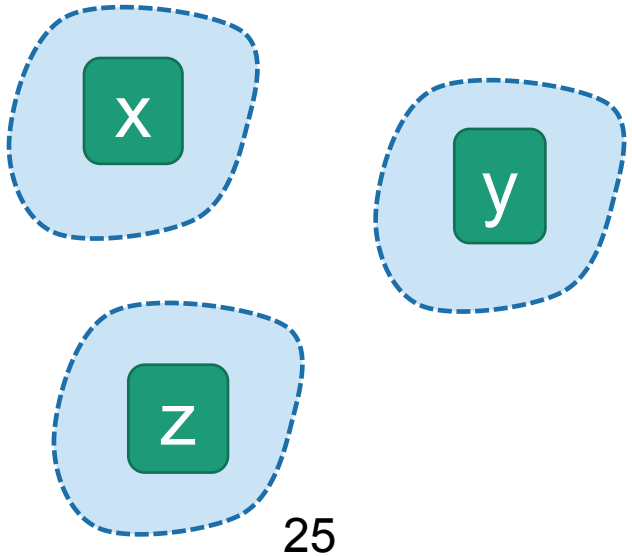
"treehouse"?

Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

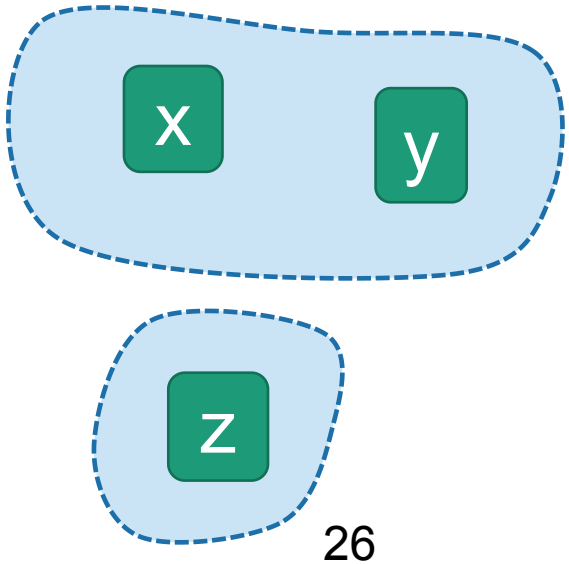


Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)



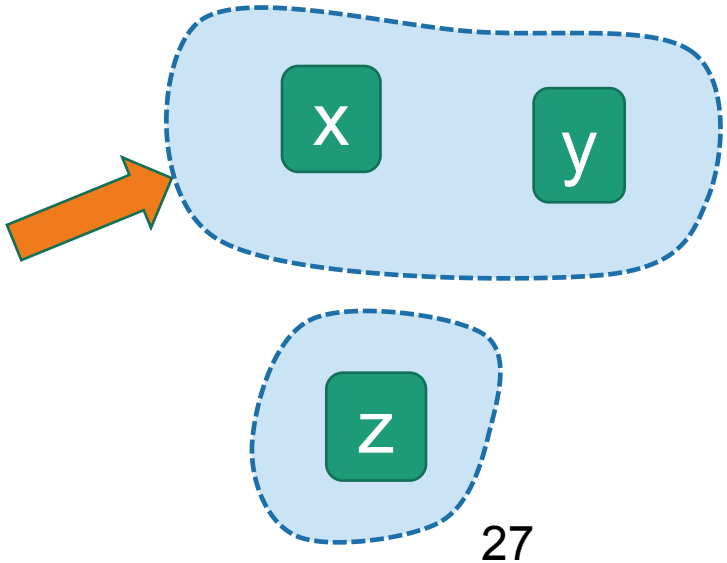
Union-find data structure
also called disjoint-set data structure

- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

makeSet(x)
makeSet(y)
makeSet(z)

union(x,y)

find(x)

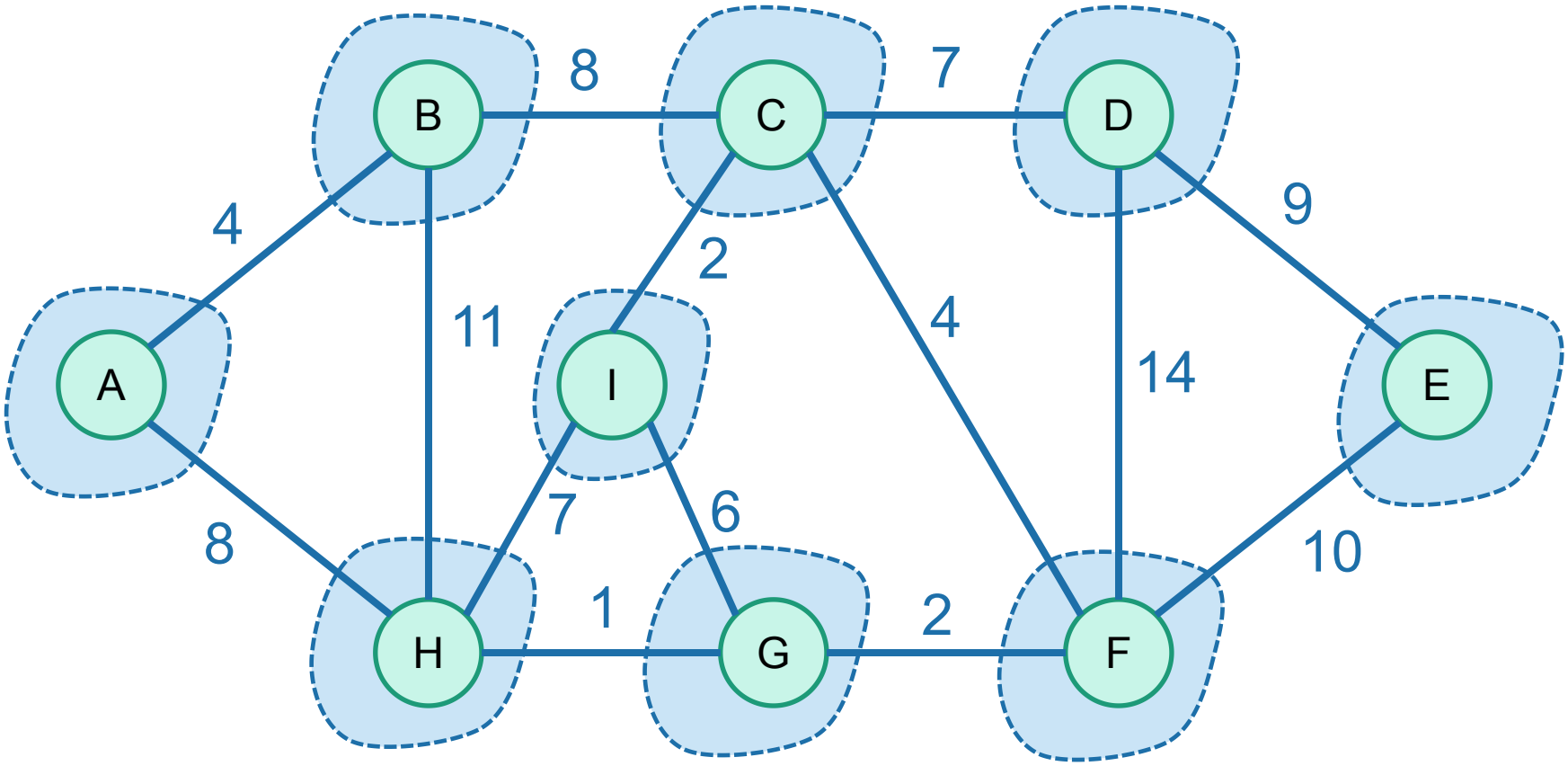


Kruskal pseudo-code

- **kruskal**(G = (V,E)):
  - Sort E by weight in non-decreasing order
  - MST = {} // initialize an empty tree
  - **for** v in V:
    - **makeSet**(v) // put each vertex in its own tree in the forest
  - **for** (u,v) in E: // go through the edges in sorted order
    - **if find**(u) != **find**(v): // if u and v are not in the same tree
      - add (u,v) to MST
      - **union**(u,v) // merge u's tree with v's tree
  - **return** MST

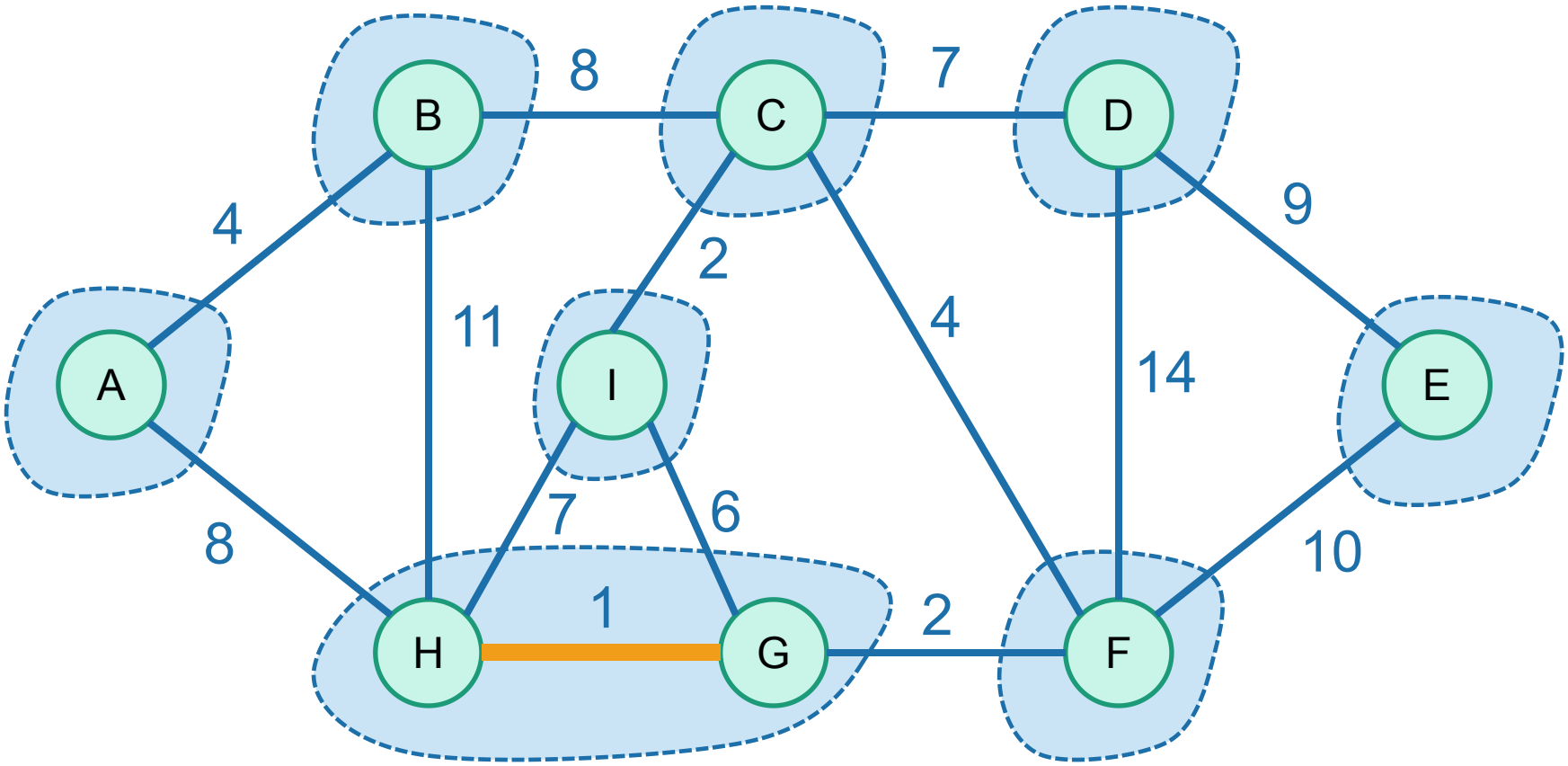
Once more…

To start, every vertex is in its own tree.



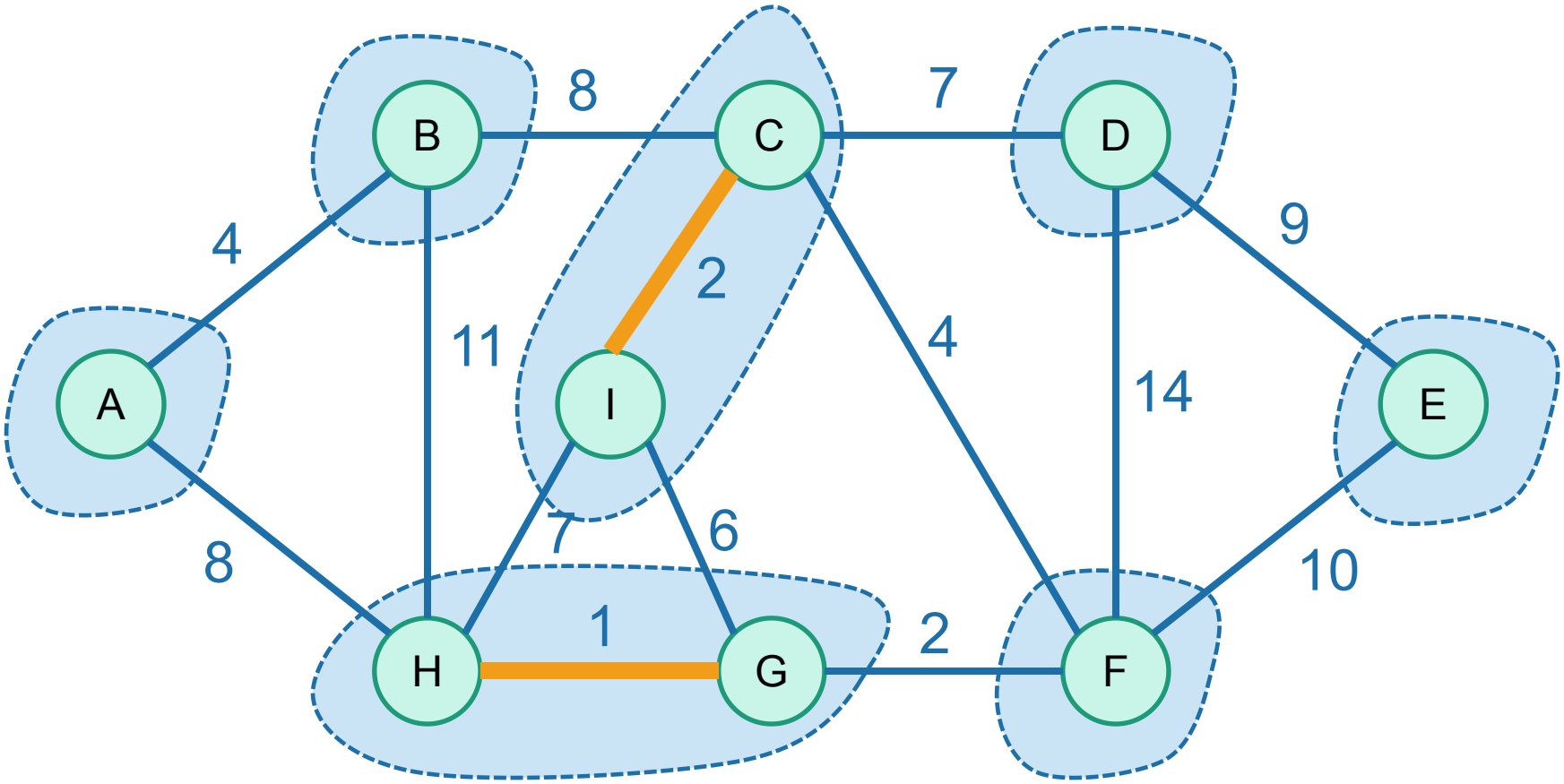
Once more…

Then start merging.



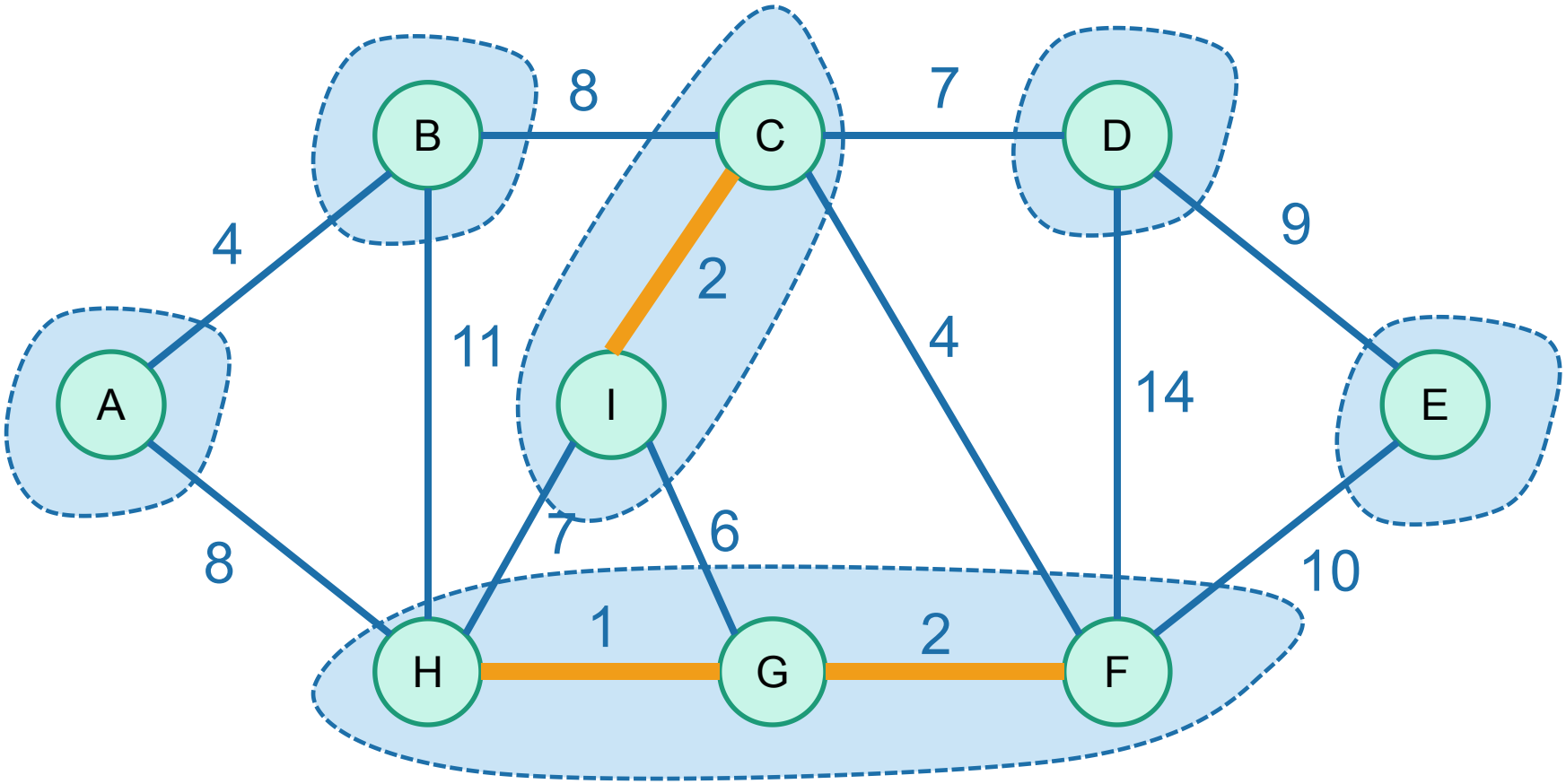
Once more…

Then start merging.



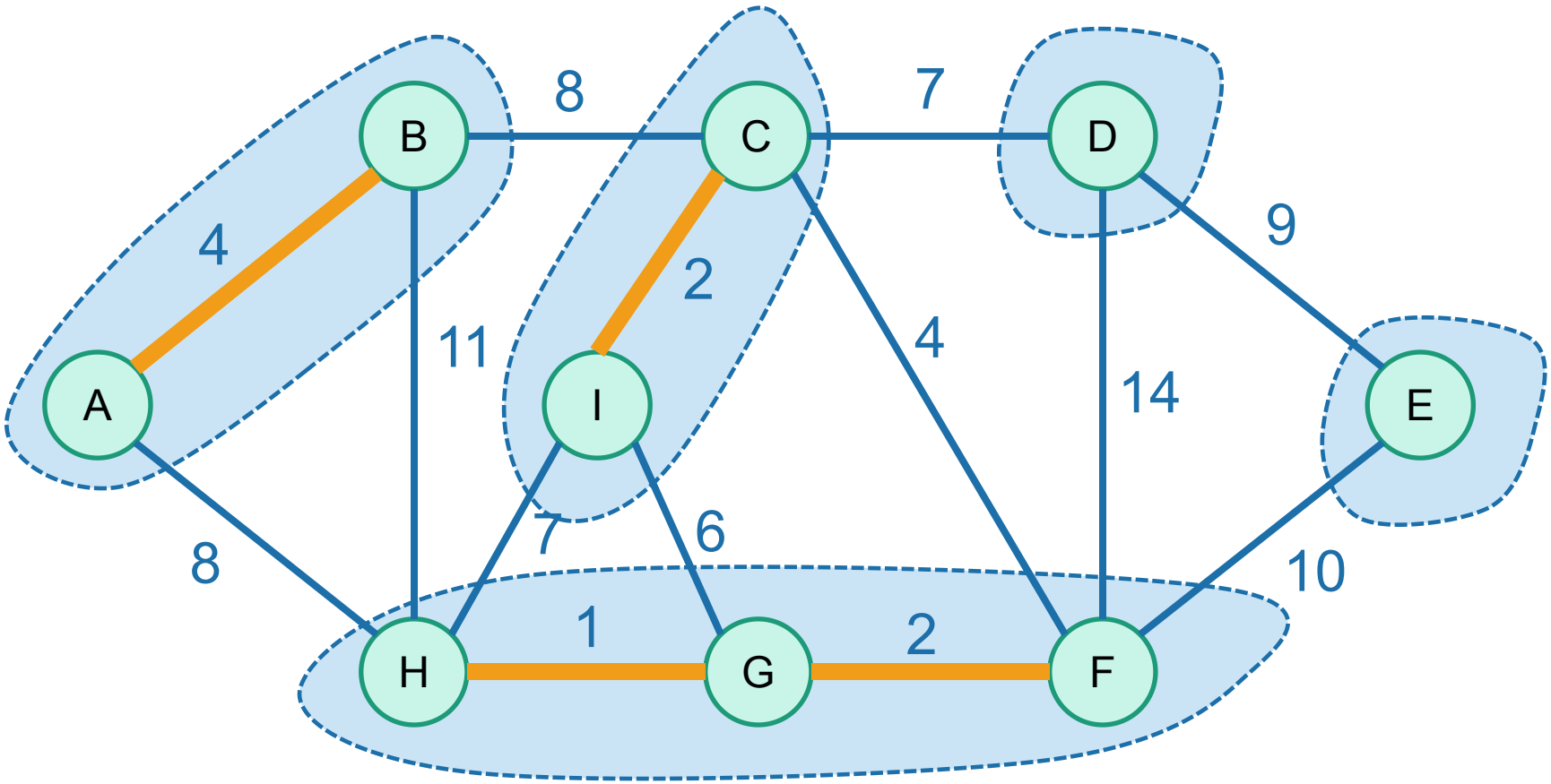
Once more…

Then start merging.



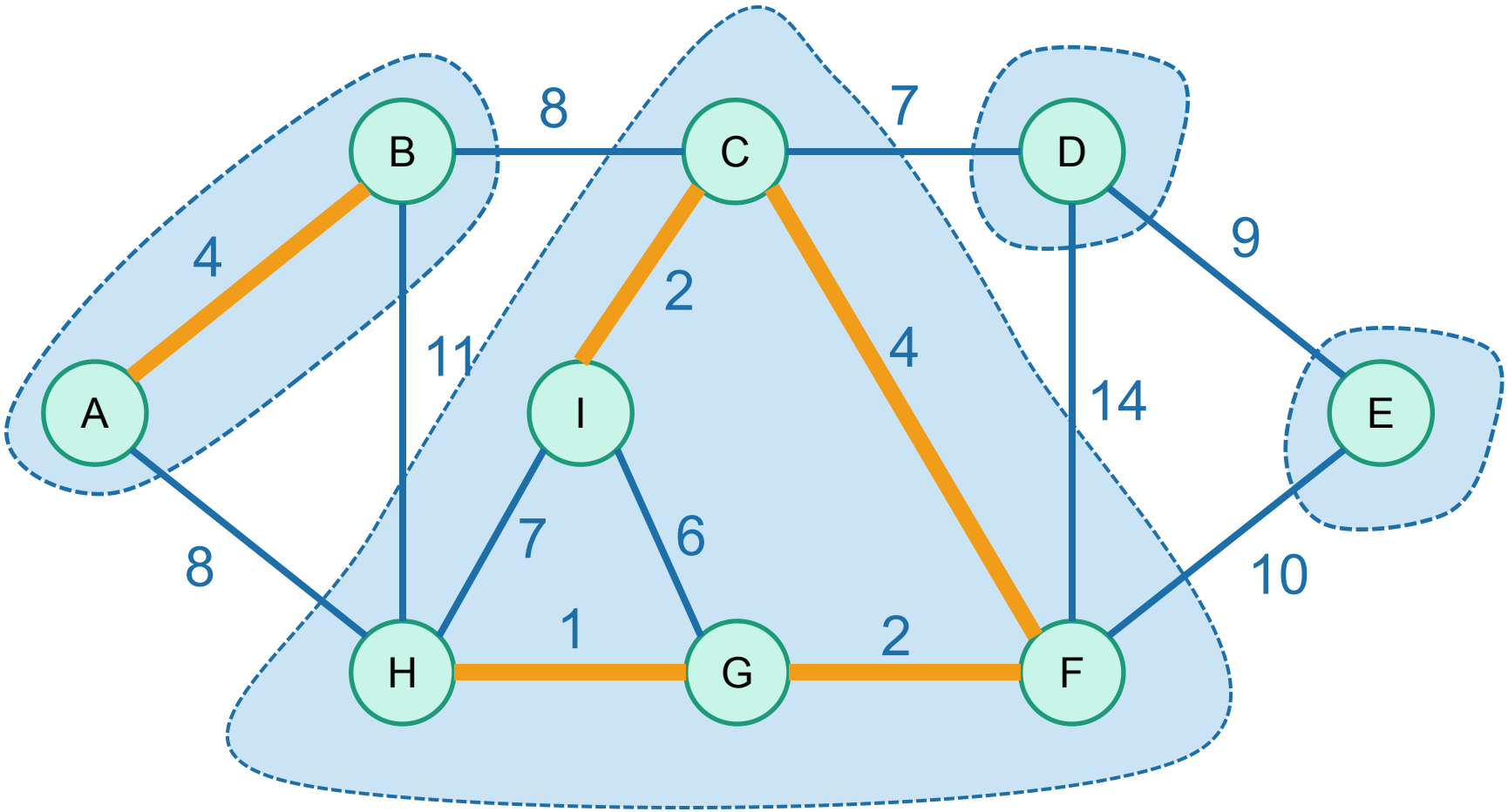
Once more…

Then start merging.



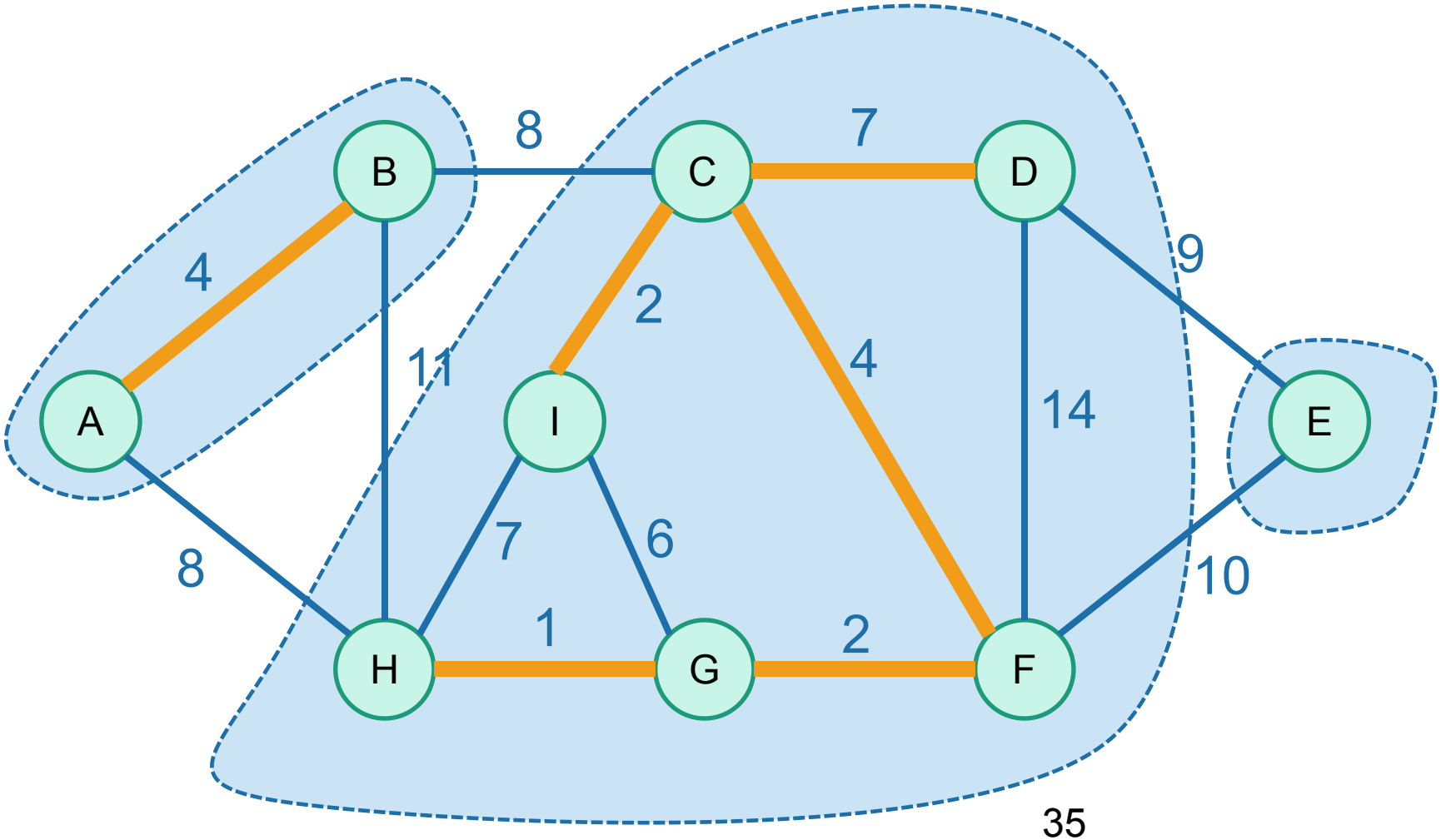
Once more…

Then start merging.



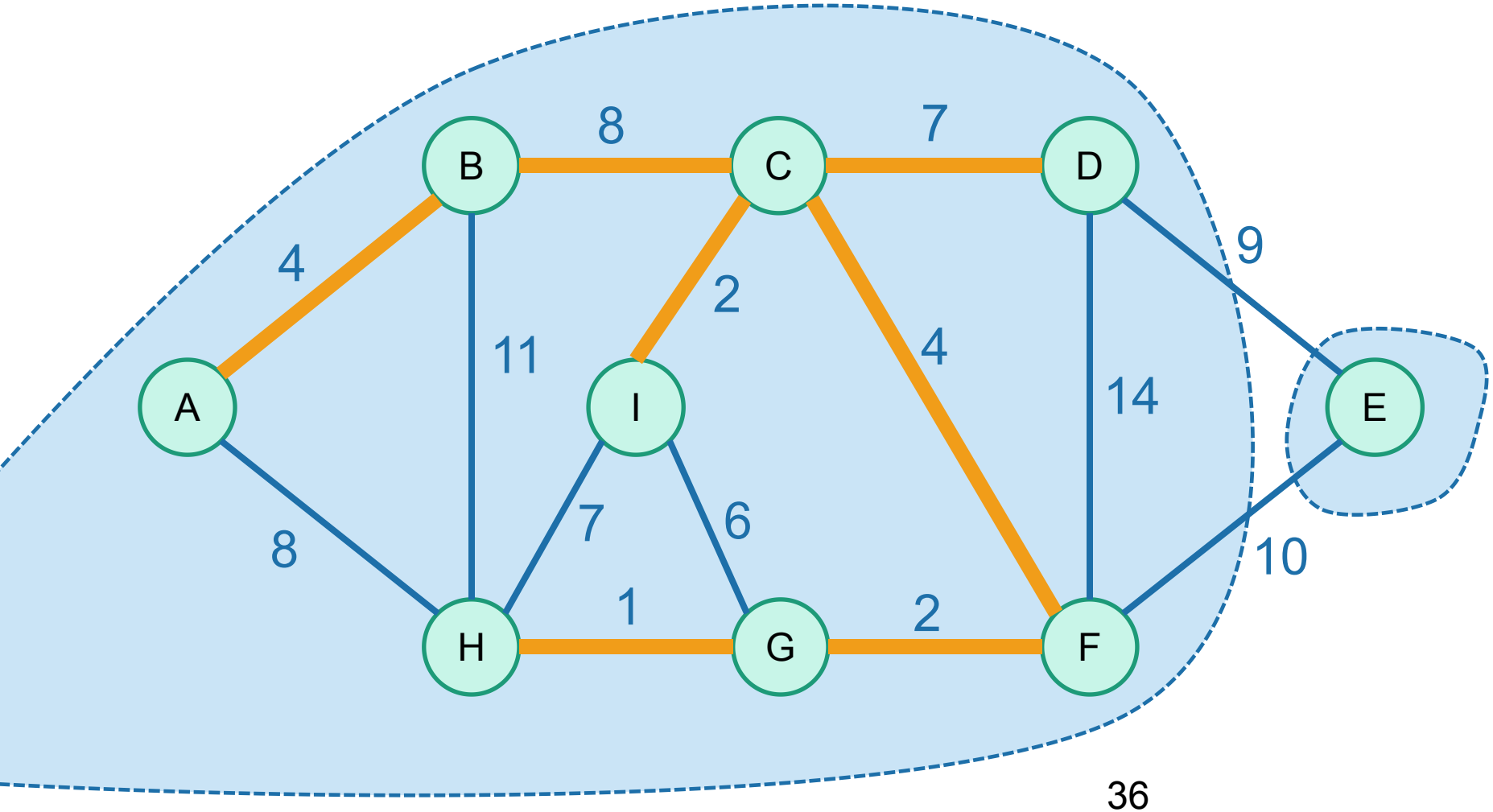
Once more…

Then start merging.



Once more…

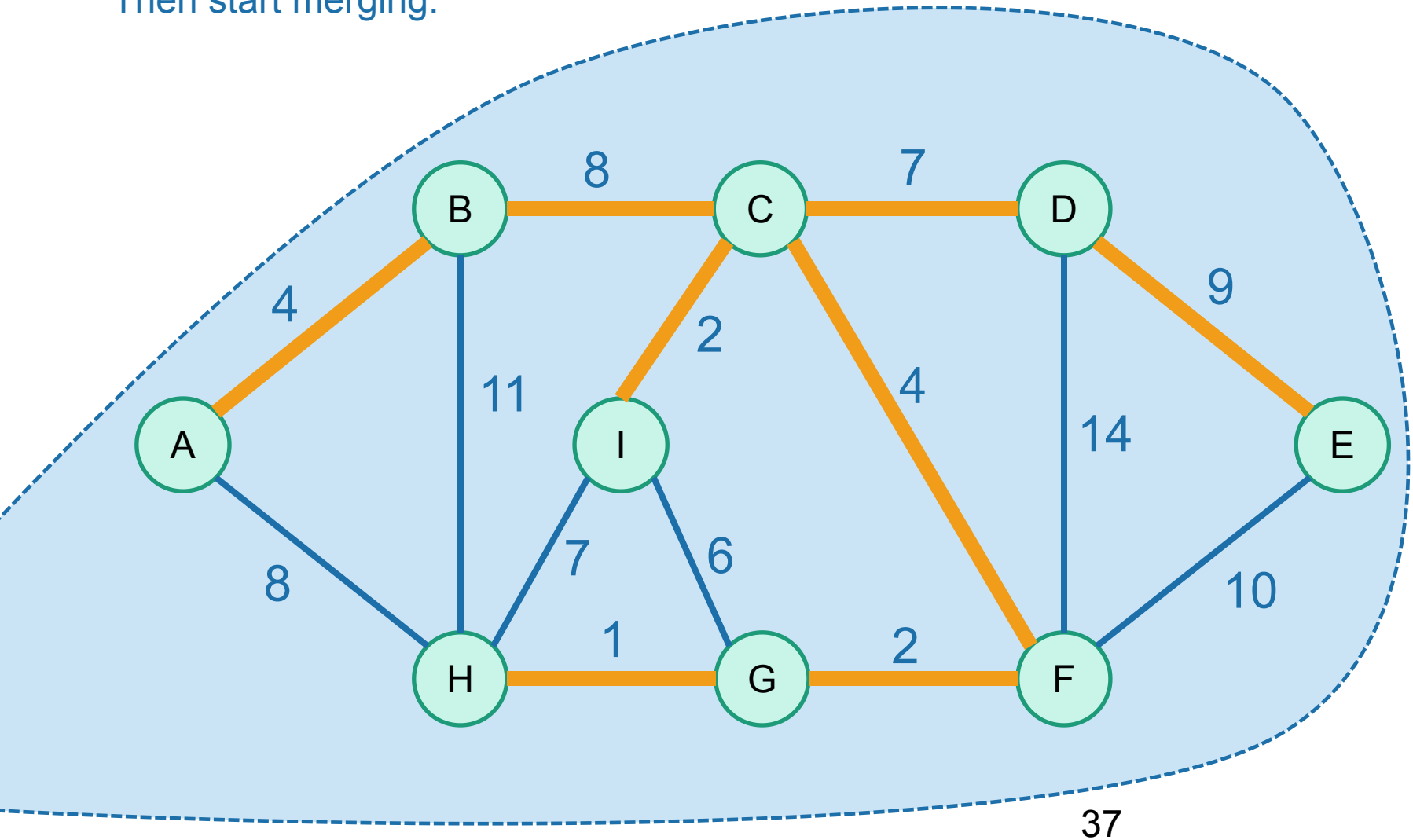
Then start merging.



Once more…

Stop when we have one big tree!

Then start merging.



Running time

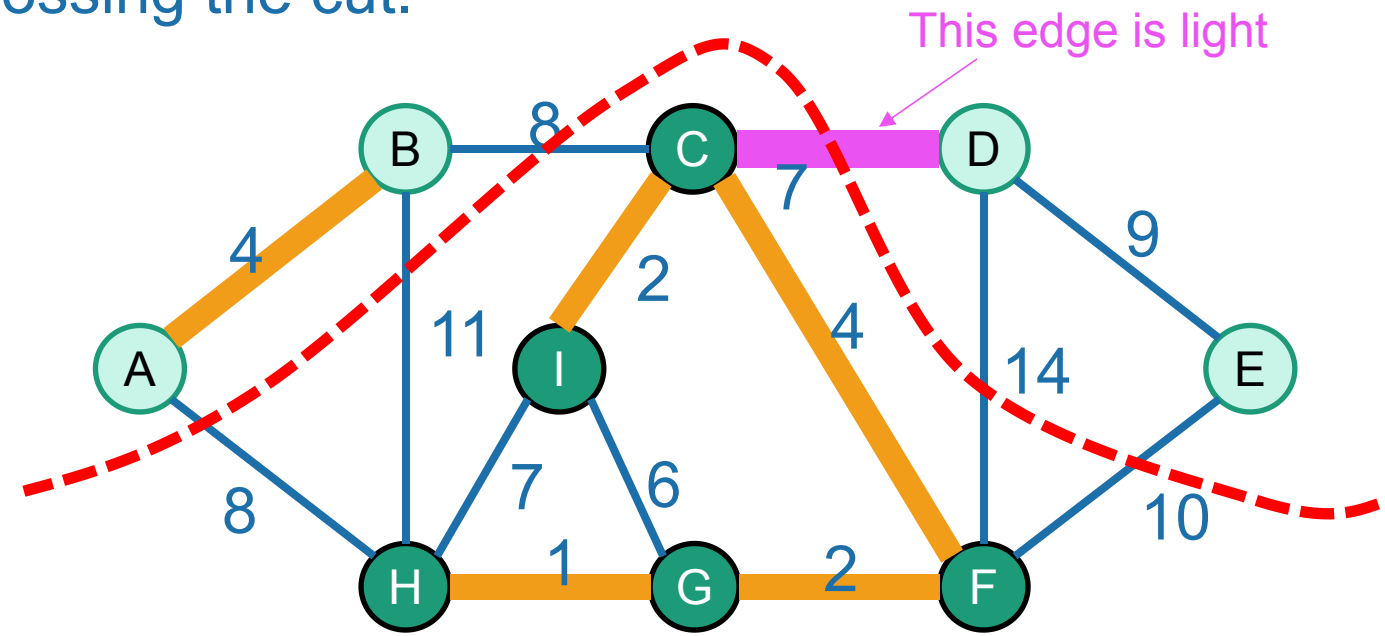
- Sorting the edges takes $O(m \log(n))$
  - In practice, if the weights are small integers we can use radixSort and take time $O(m)$
- For the rest:
  - n calls to **makeSet**
    - put each vertex in its own set
  - $2m$ calls to **find**
    - for each edge, **find** its endpoints
  - n calls to **union**
    - we will never add more than $n - 1$ edges to the tree,
    - so we will never call **union** more than $n - 1$ times.
- Total running time:
  - Worst-case $O(mlog(n))$, just like Prim with an RBtree.

In practice, each of makeSet, find, and union run in constant time*

*technically, they run in *amortized time* $O(\alpha(n))$, where $\alpha(n)$ is the *inverse Ackermann function*. $\alpha(n) \leq 4$ provided that n is smaller than the number of atoms in the universe.

Both Prim and Kruskal

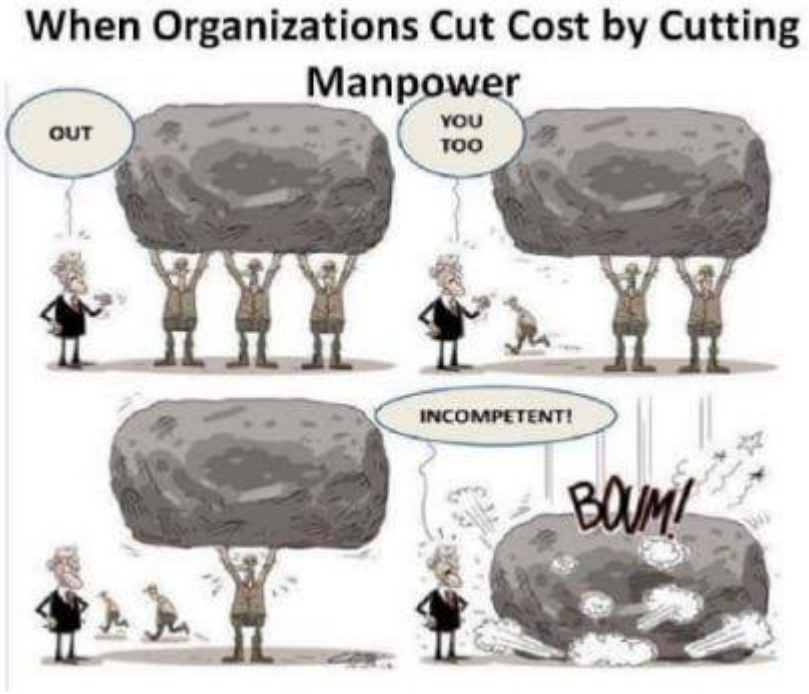
- Greedy algorithms for MST.
- Similar reasoning:
  - Optimal substructure: subgraphs generated by cuts.
  - The way to make safe choices is to choose light edges crossing the cut.



S is the set of **thick orange** edges

Today

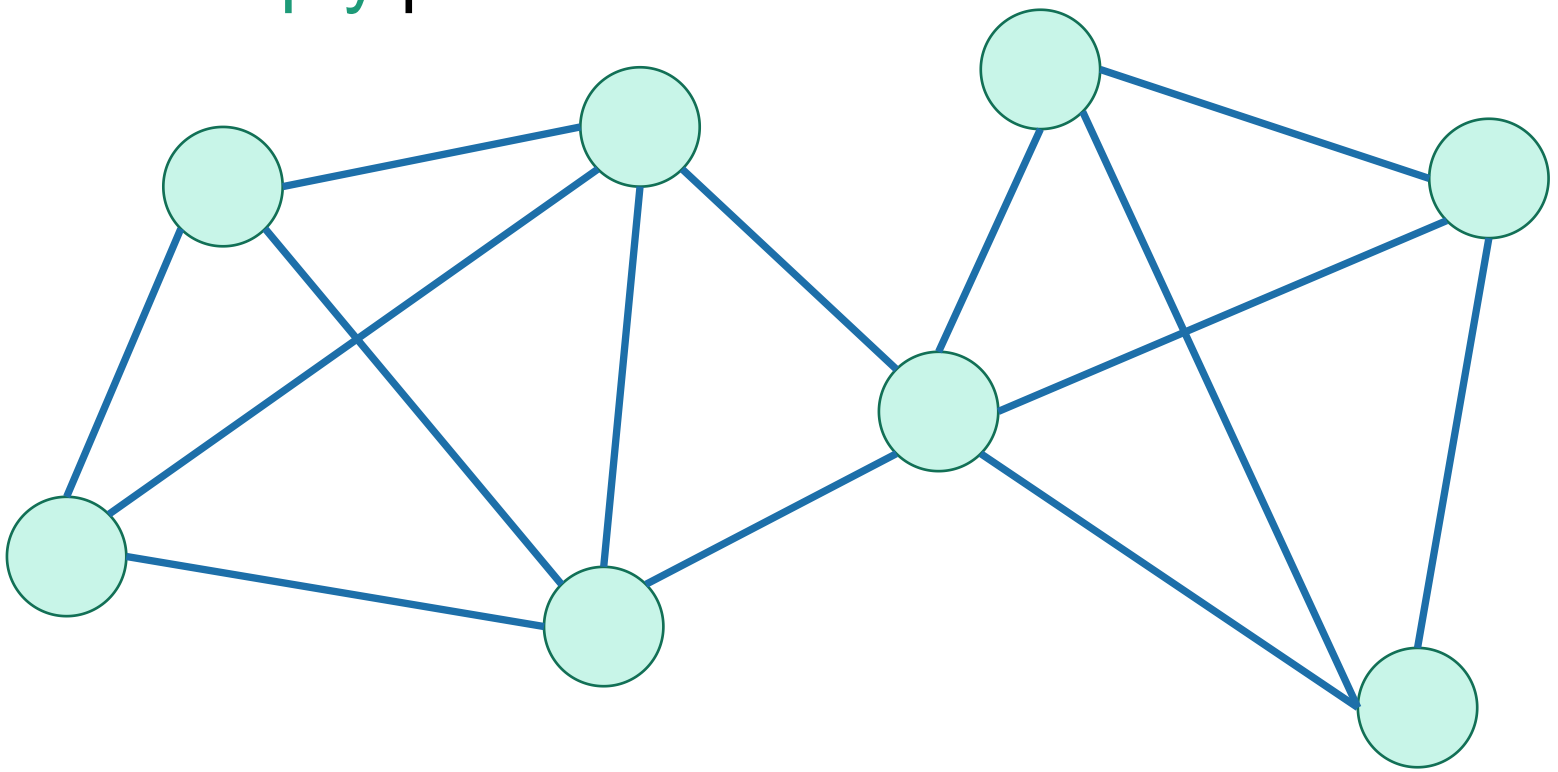
- Cuts anf Flow



Recall: cuts in graphs

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

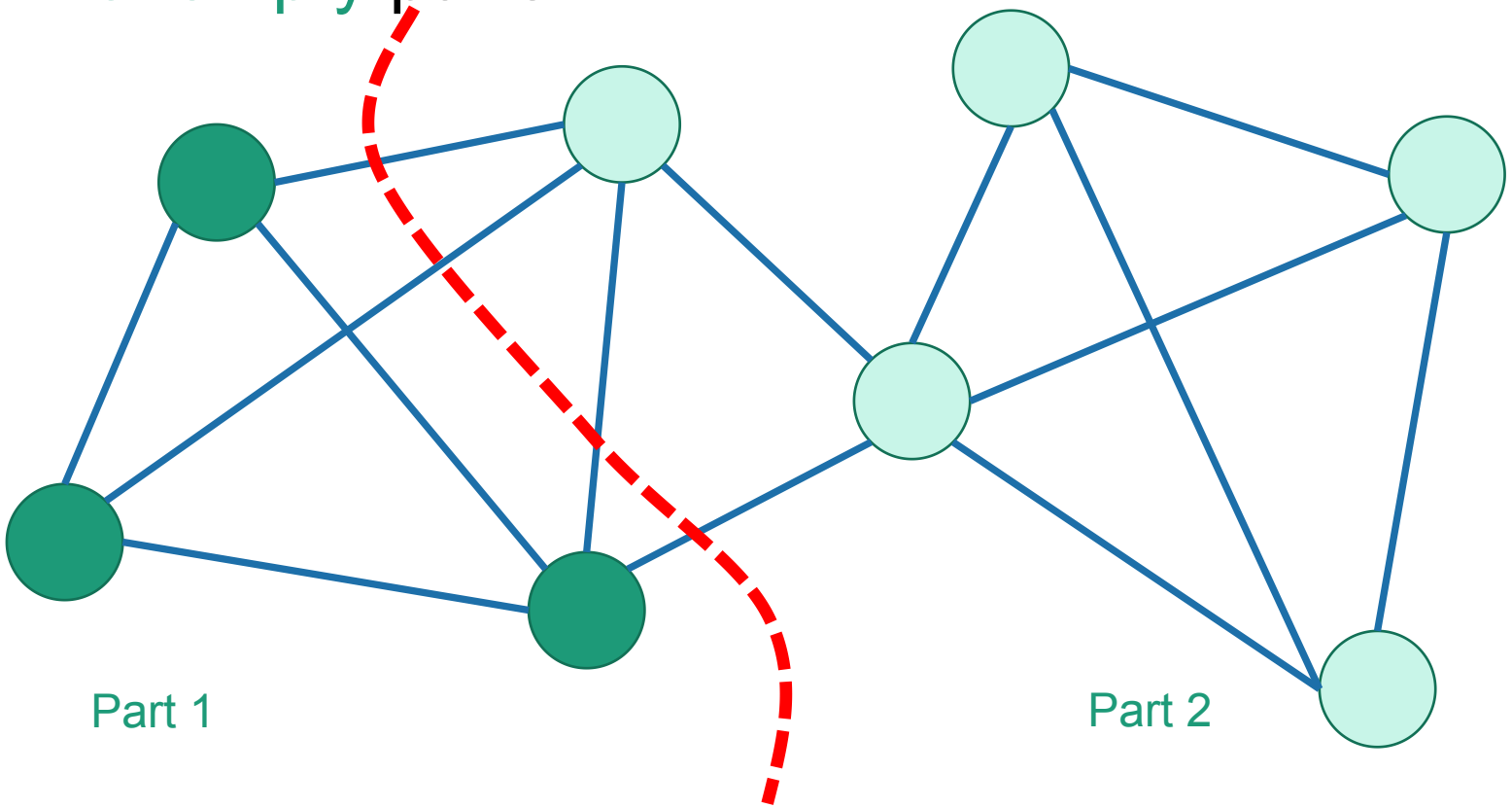
- A cut is a partition of the vertices into two nonempty parts.



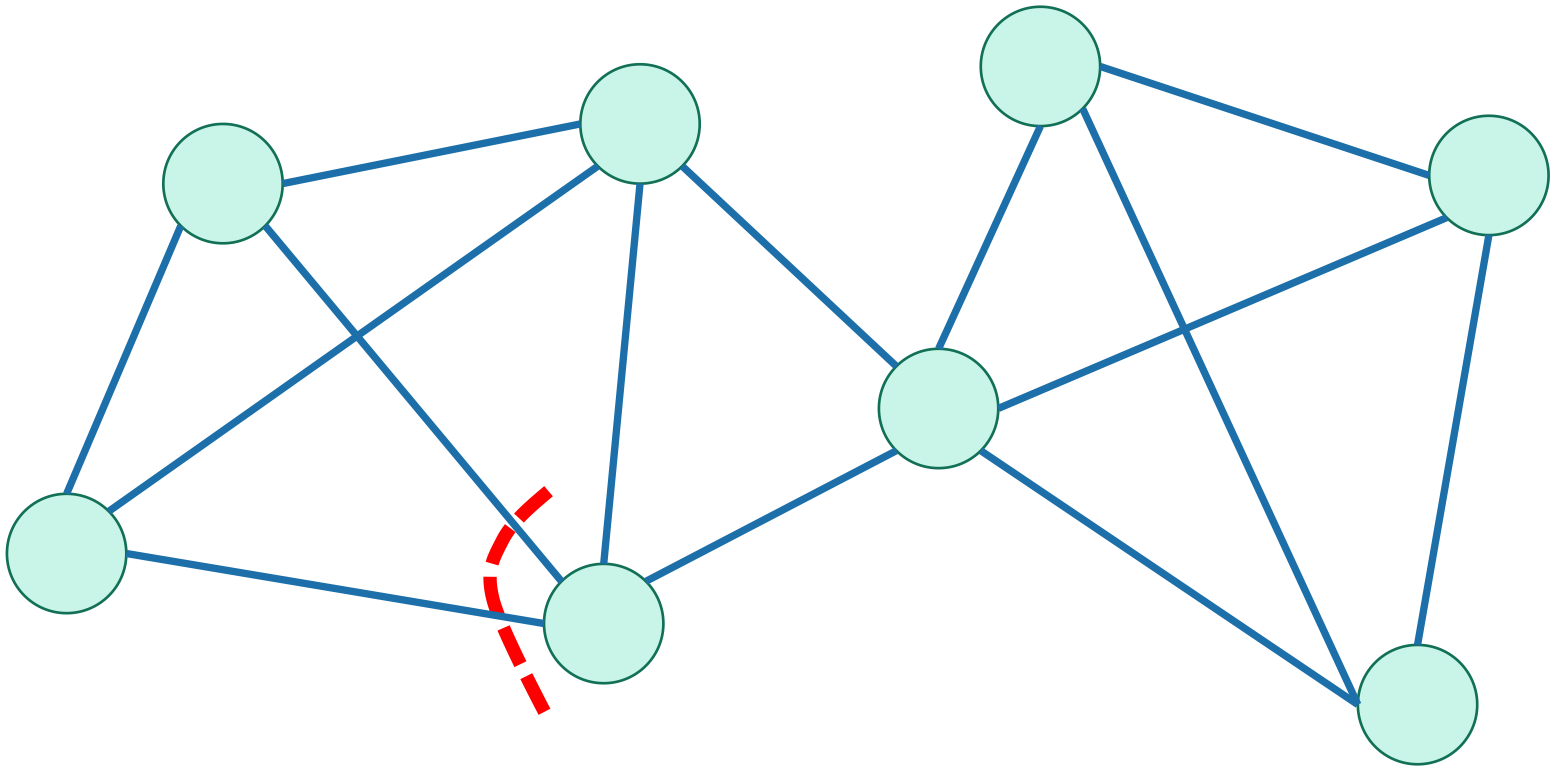
Recall: cuts in graphs

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

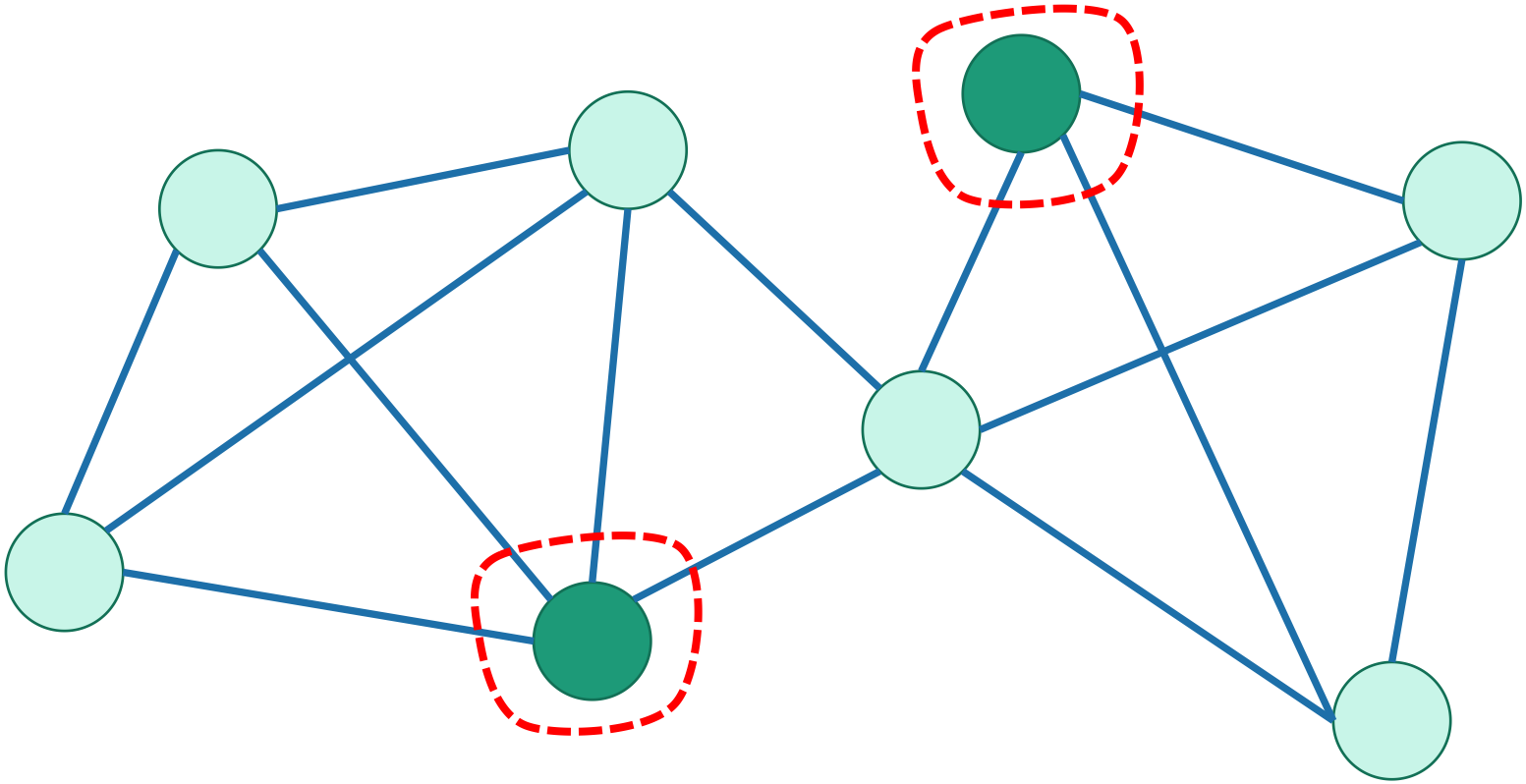
- A cut is a partition of the vertices into two nonempty parts.



This is not a cut



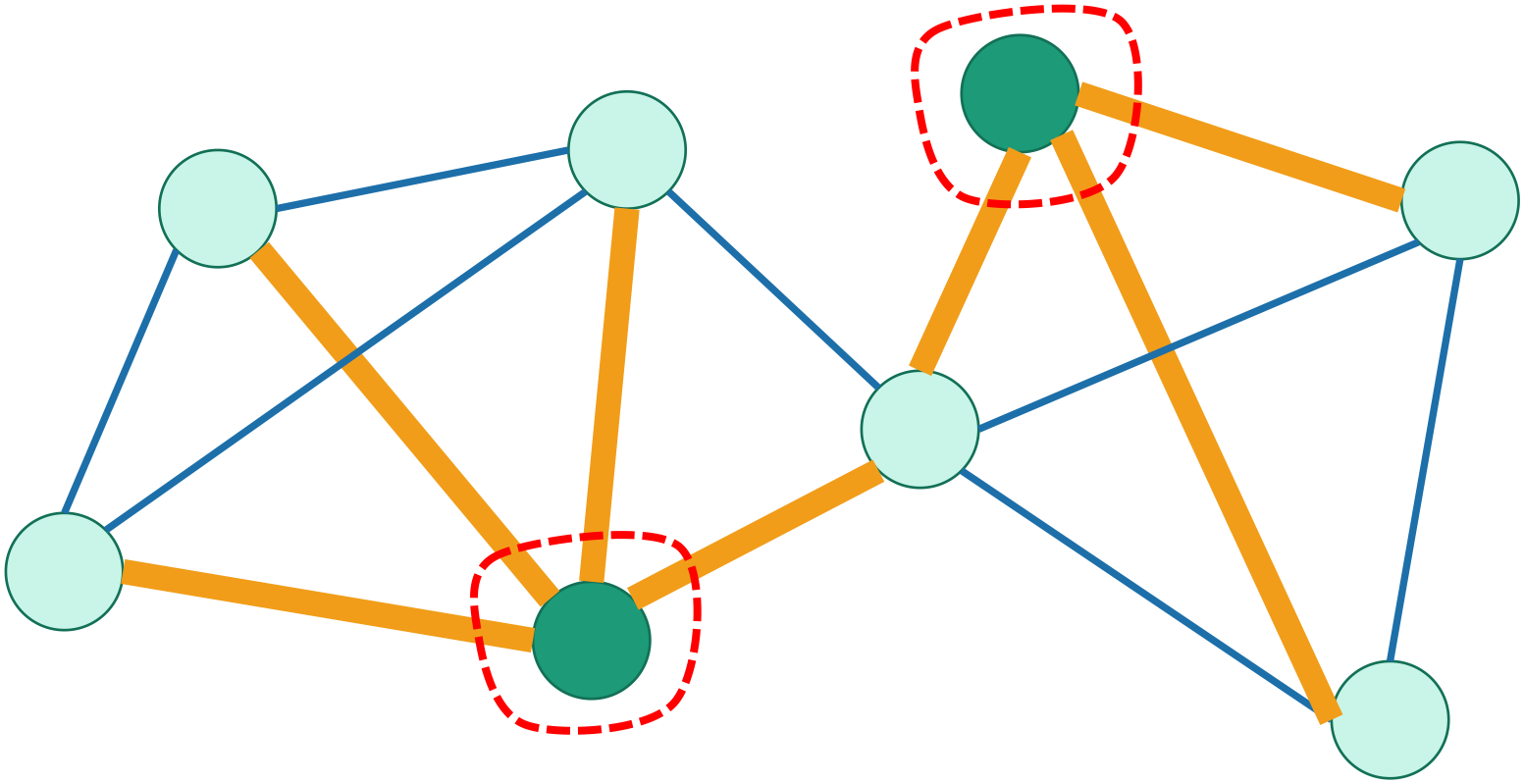
This is a cut



This is a cut

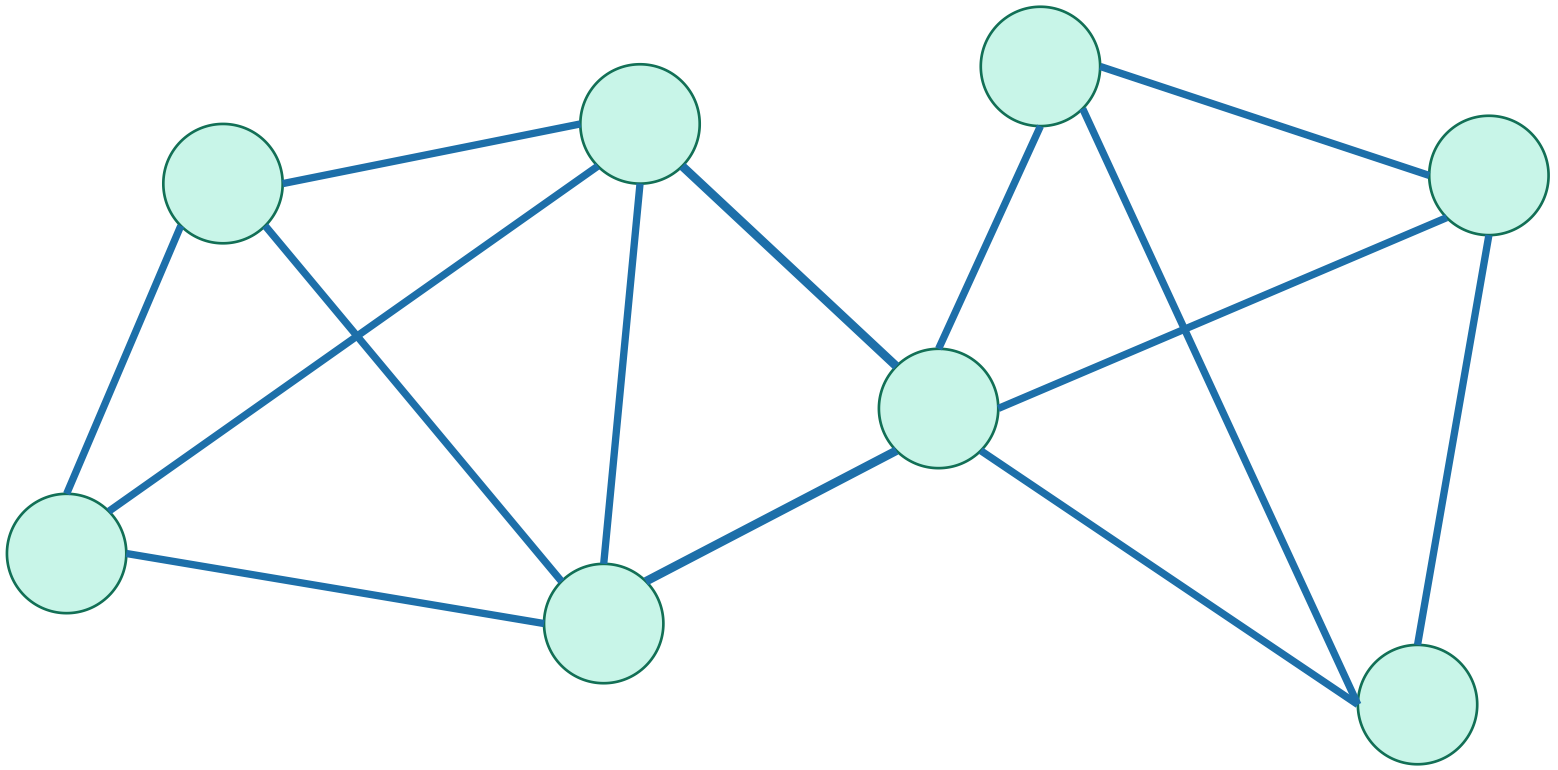
These edges **cross the cut.**

- They go from one part to the other



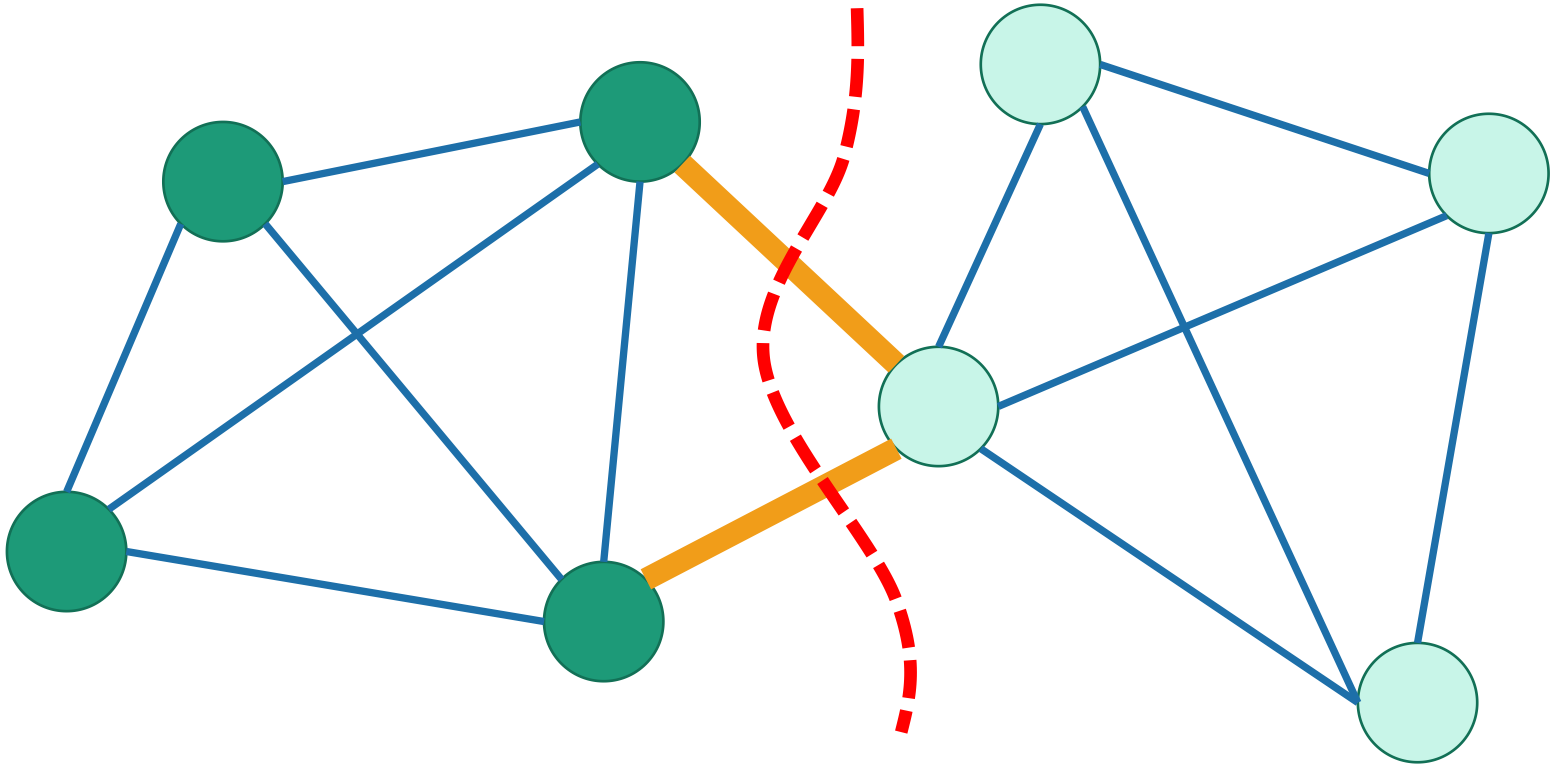
A (global) minimum cut

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



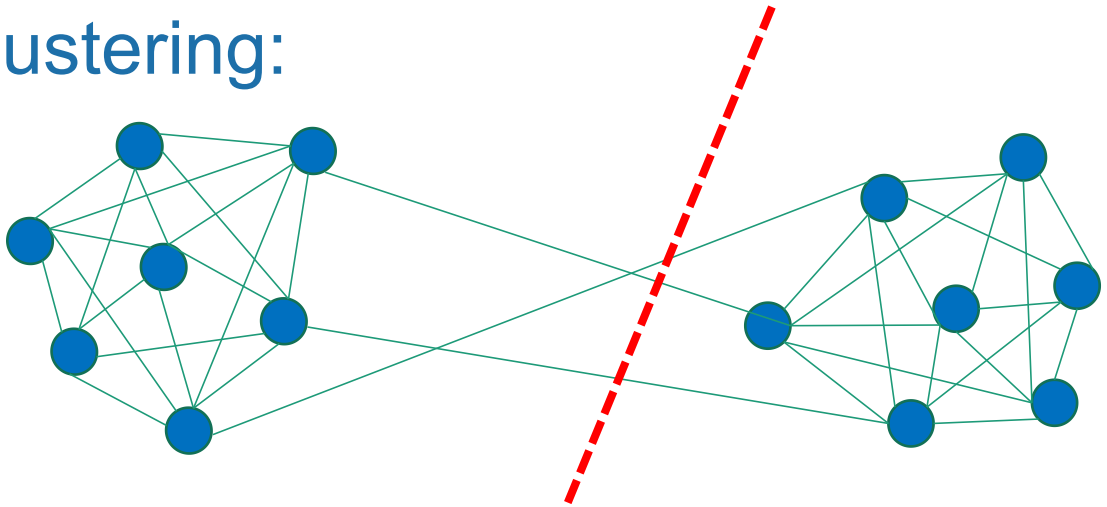
A (global) minimum cut

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

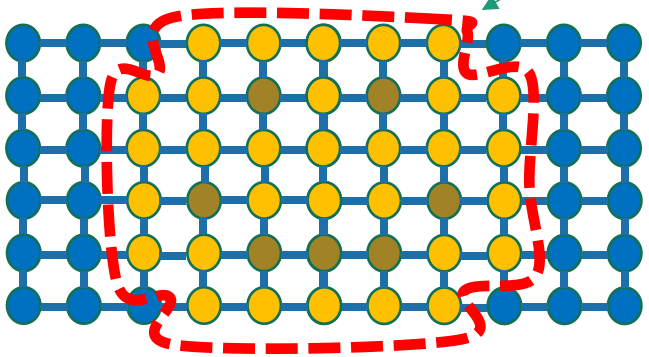


Why might we care about global minimum cuts?

- Clustering:



- Image Segmentation



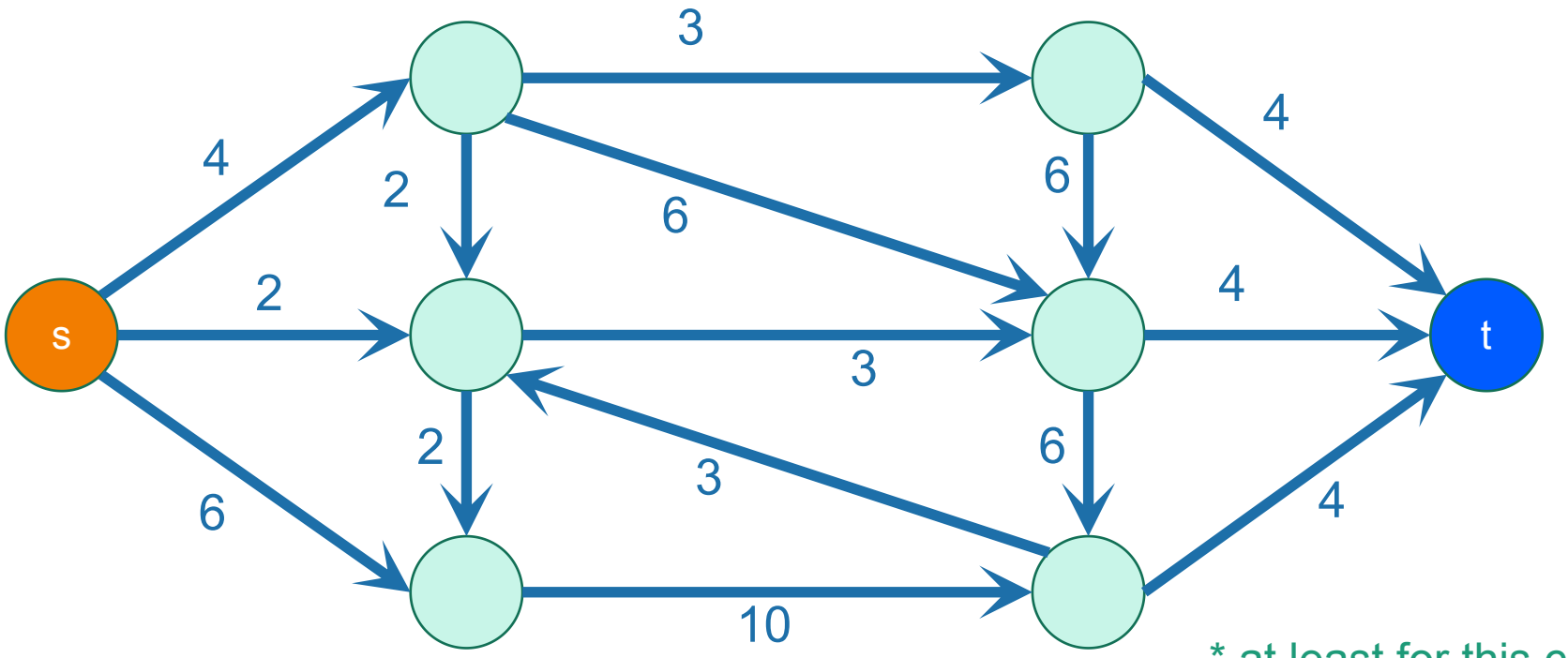
*For the rest of today edges aren't weighted; but the algorithm can be adapted to deal with edge weights.

Karger's algorithm (Please Read it. Not Covered Today)

- Finds **global minimum cuts** in undirected graphs
- Randomized algorithm
  - But a different sort of randomized algorithm than Quicksort!
- Karger's algorithm **might be wrong**.
  - While 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.

s-t cut

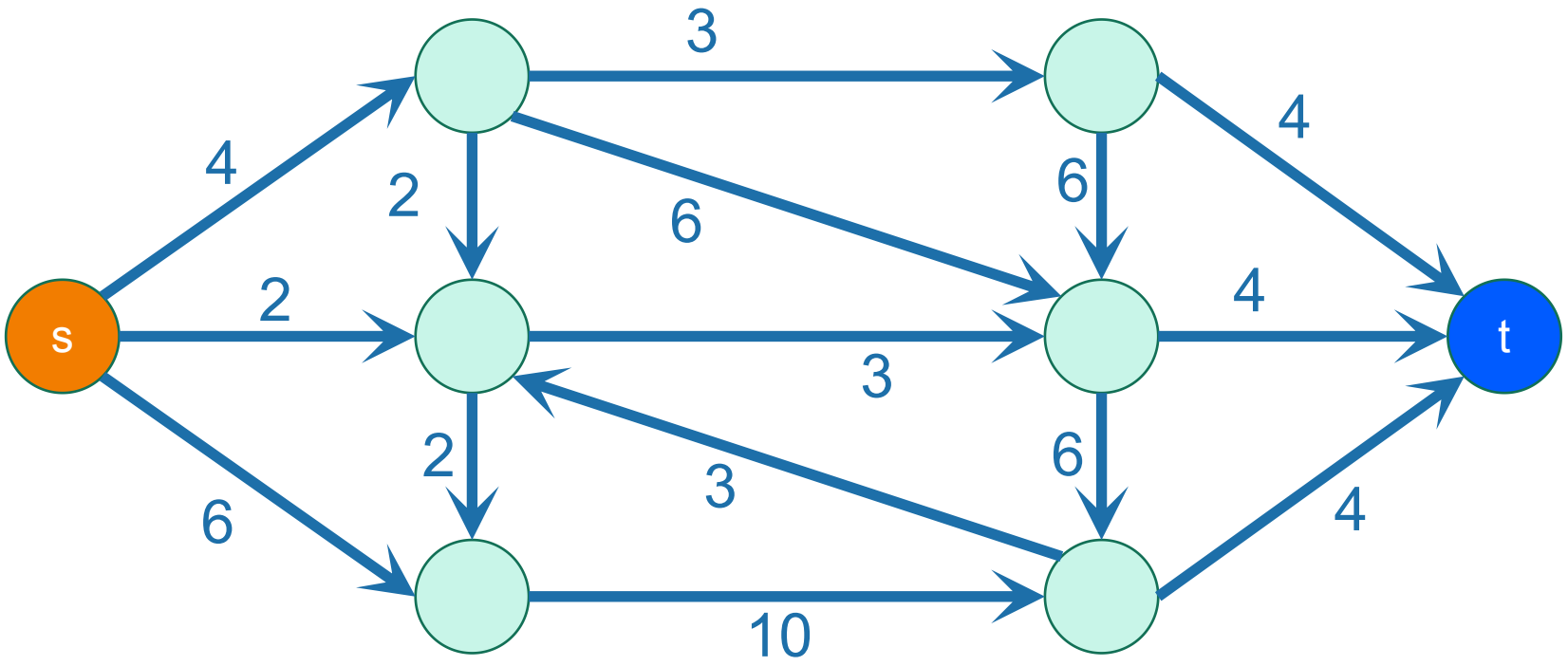
- Graphs are directed and edges have “capacities” (weights)
- We have a special “source” vertex s and “sink” vertex t.
  - s has only outgoing edges*
  - t has only incoming edges*



* at least for this class

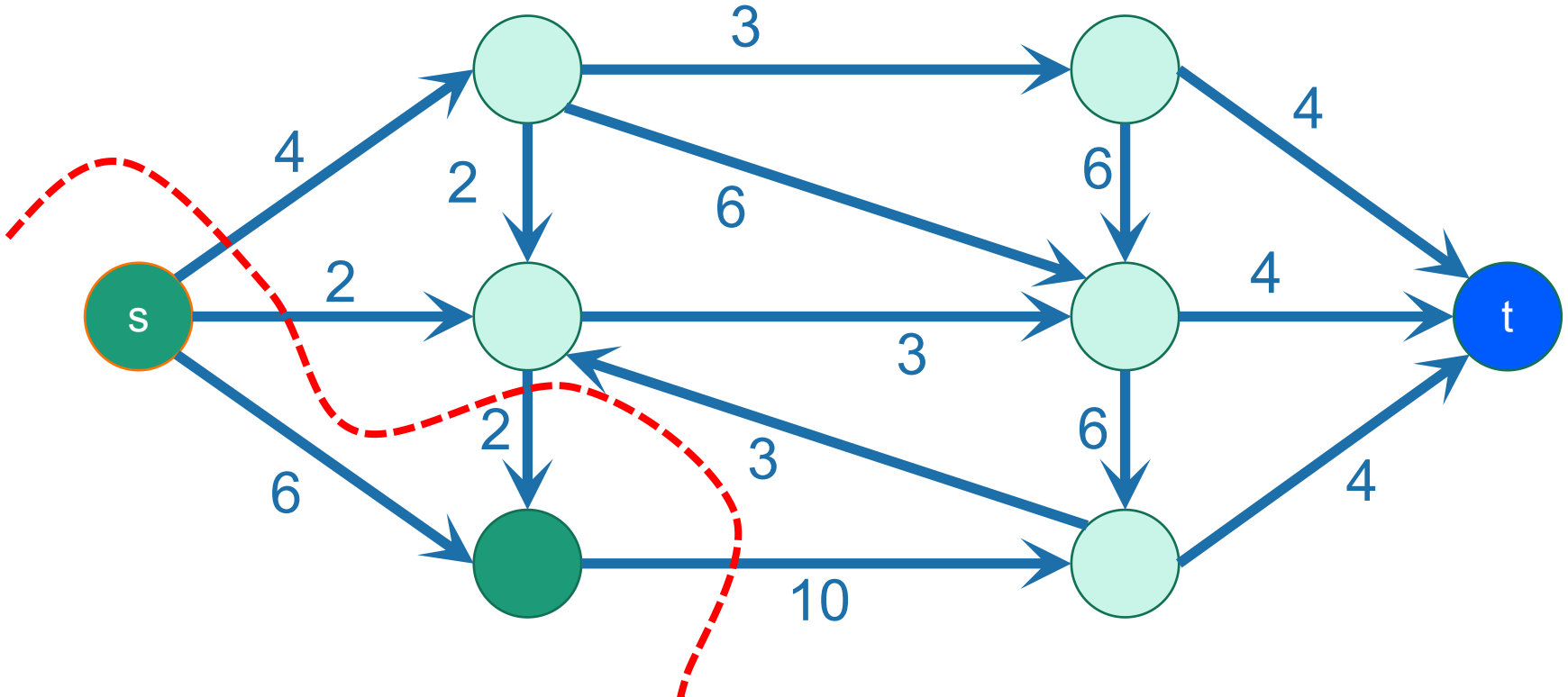
An **s-t cut**

is a cut which separates s from t



An **s-t cut**

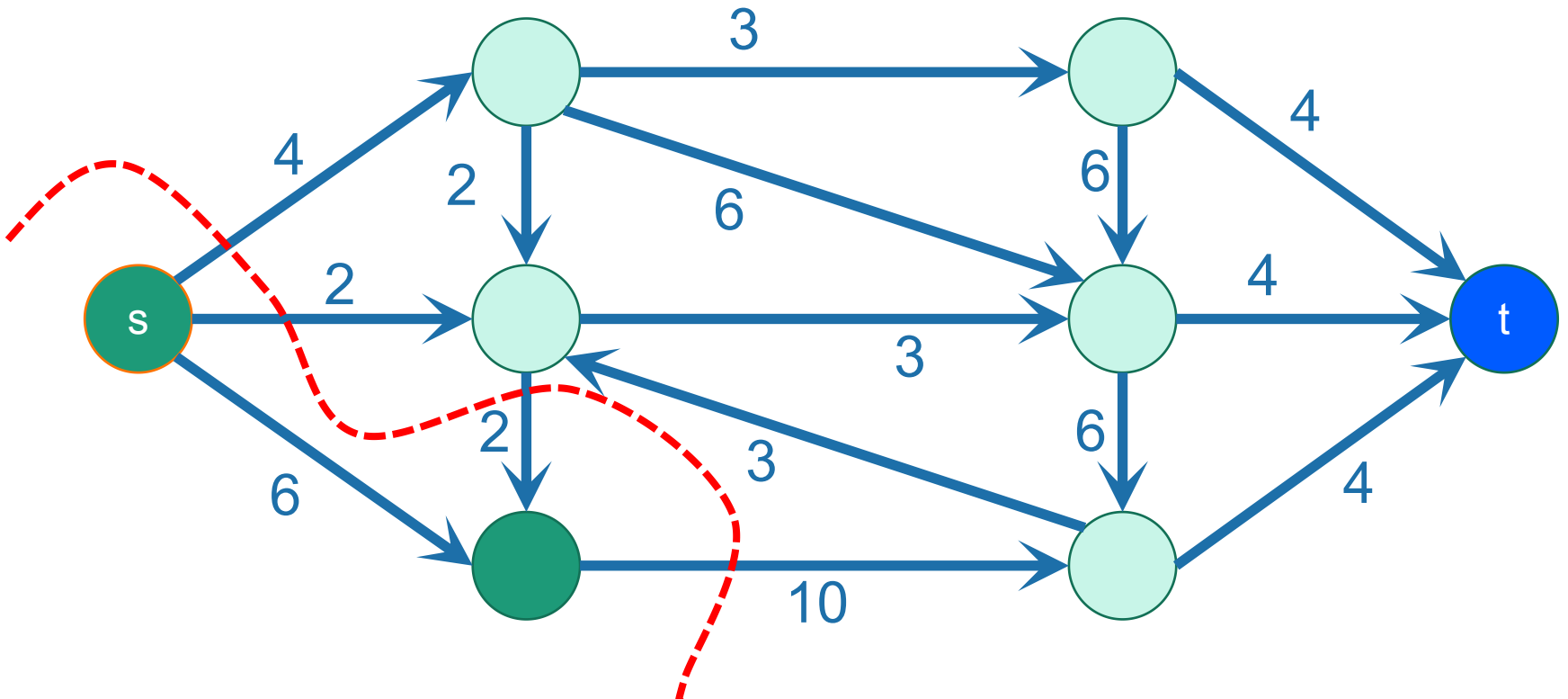
is a cut which separates s from t



An **s-t cut**

is a cut which separates s from t

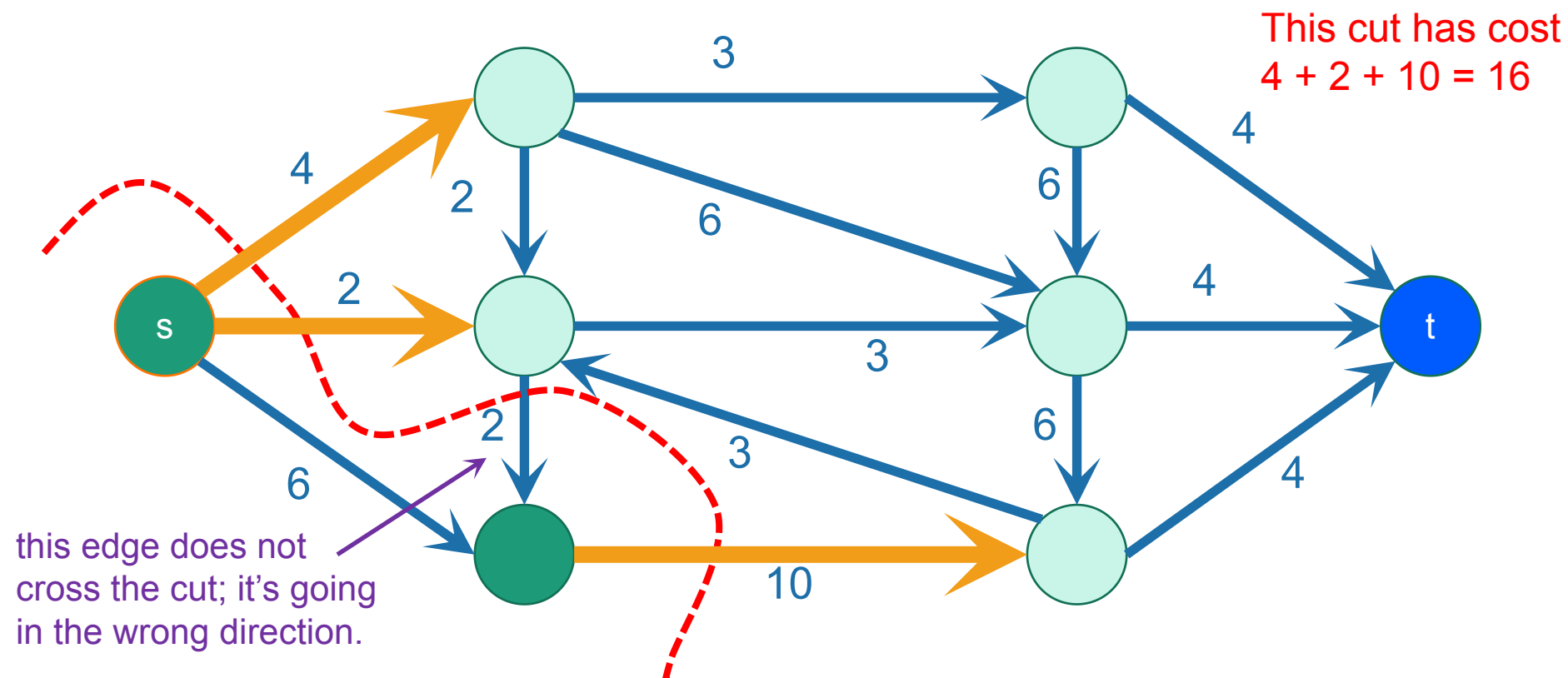
- An edge crosses the cut if it goes from s's side to t's side.



An **s-t cut**

is a cut which separates s from t

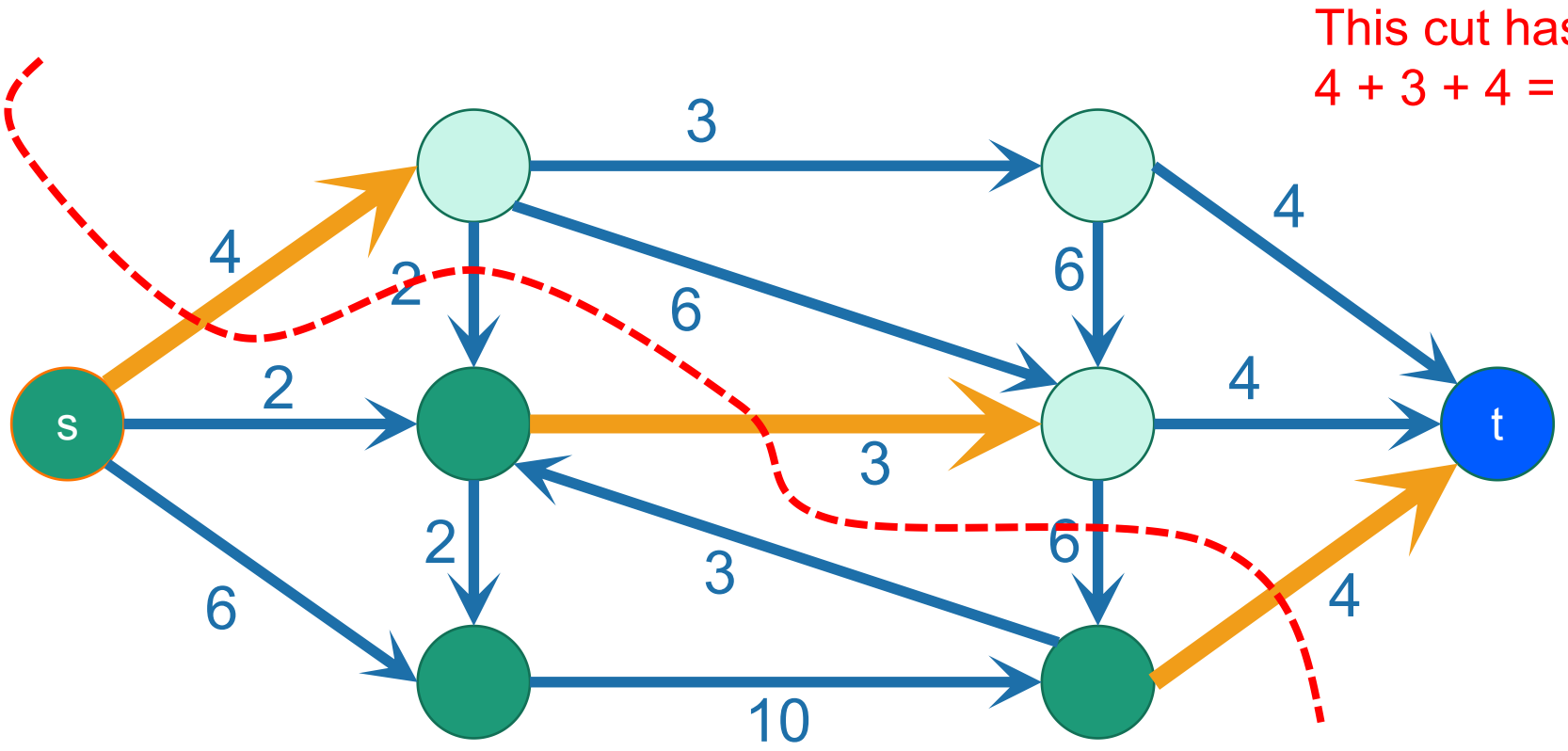
- An edge crosses the cut if it goes from s's side to t's side.
- The cost (or capacity) of a cut is the sum of the capacities of the edges that cross the cut



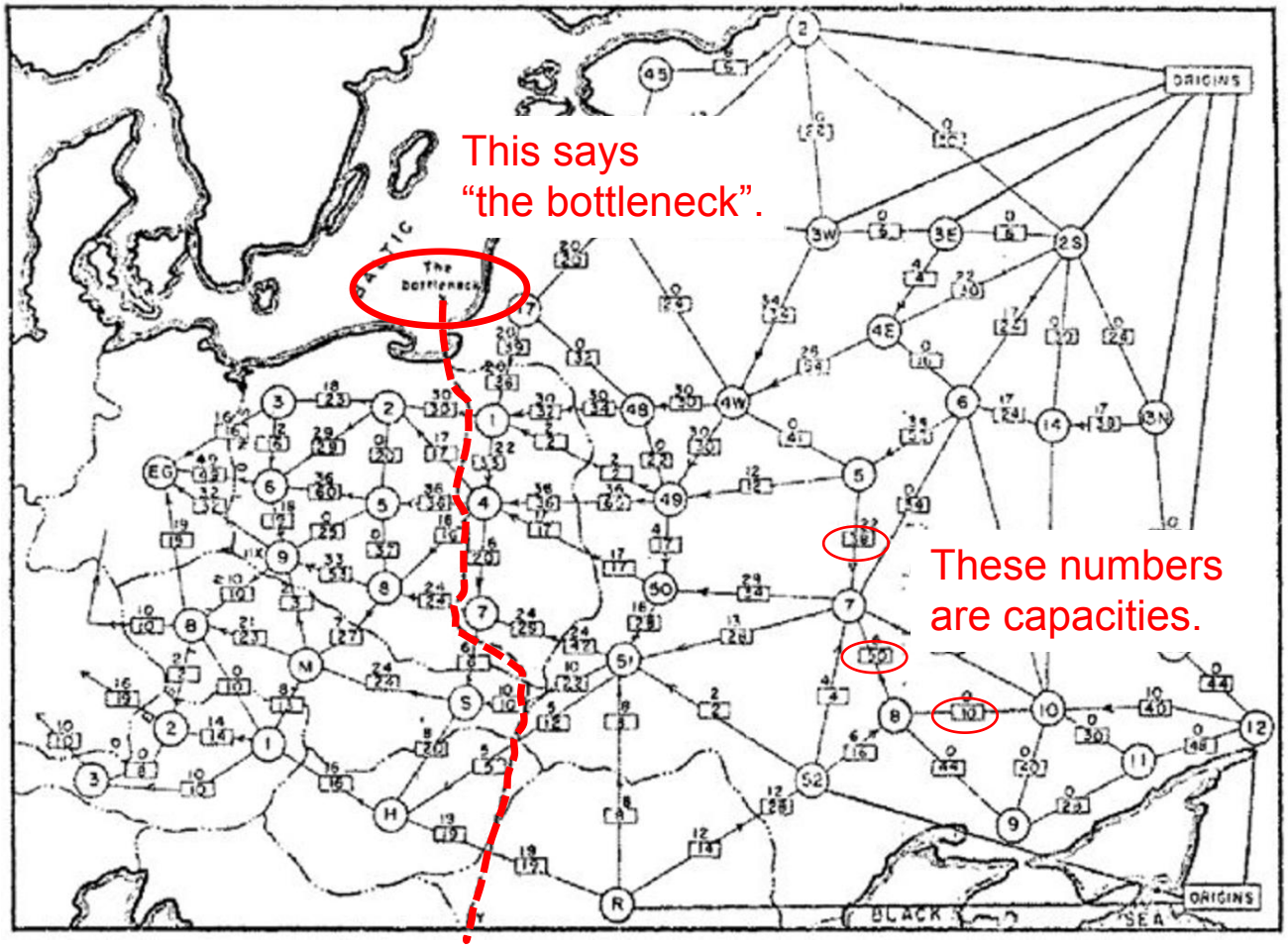
A minimum **s-t cut**

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

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



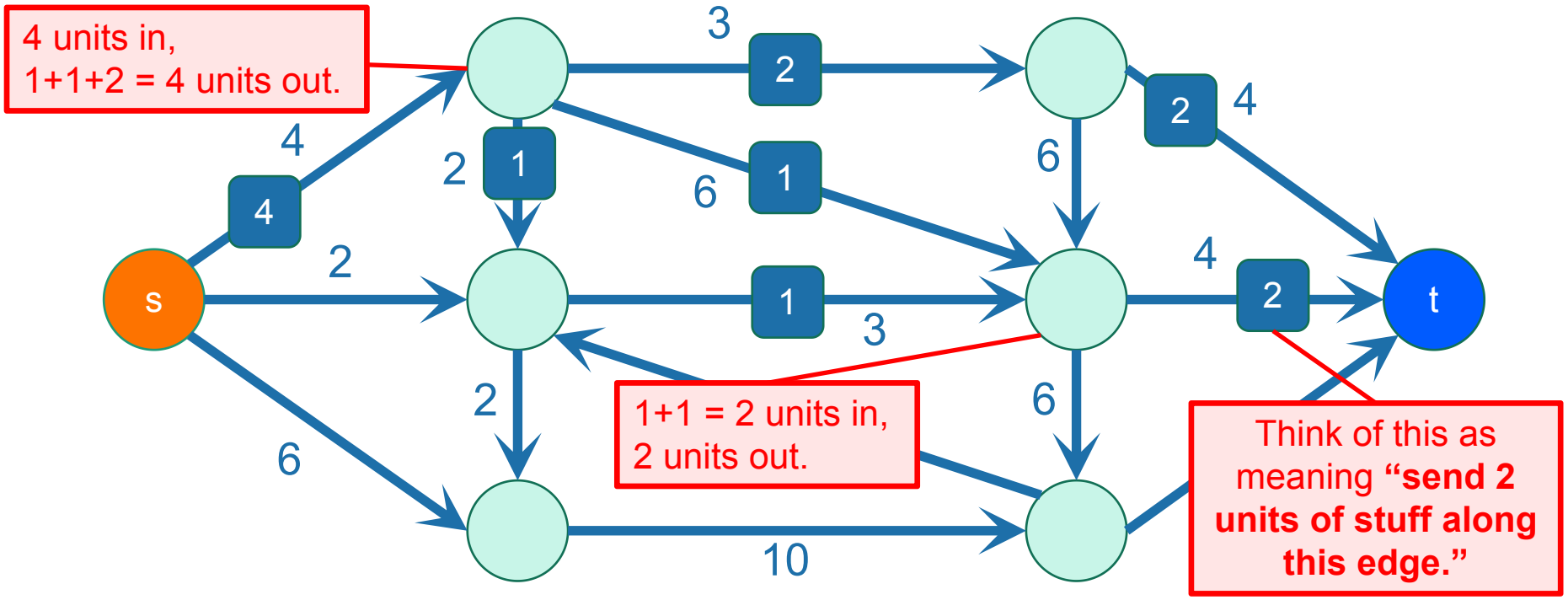
Example where this comes up



- 1955 map of rail networks from the Soviet Union to Eastern Europe.
  - Declassified in 1999.
  - 44 vertices, 105 edges
- The US wanted to cut off routes from suppliers in Russia to Eastern Europe as efficiently as possible
- In 1955, Ford and Fulkerson at the RAND corporation 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 less than its capacity.
- At each vertex, the incoming flows must equal outgoing flows.

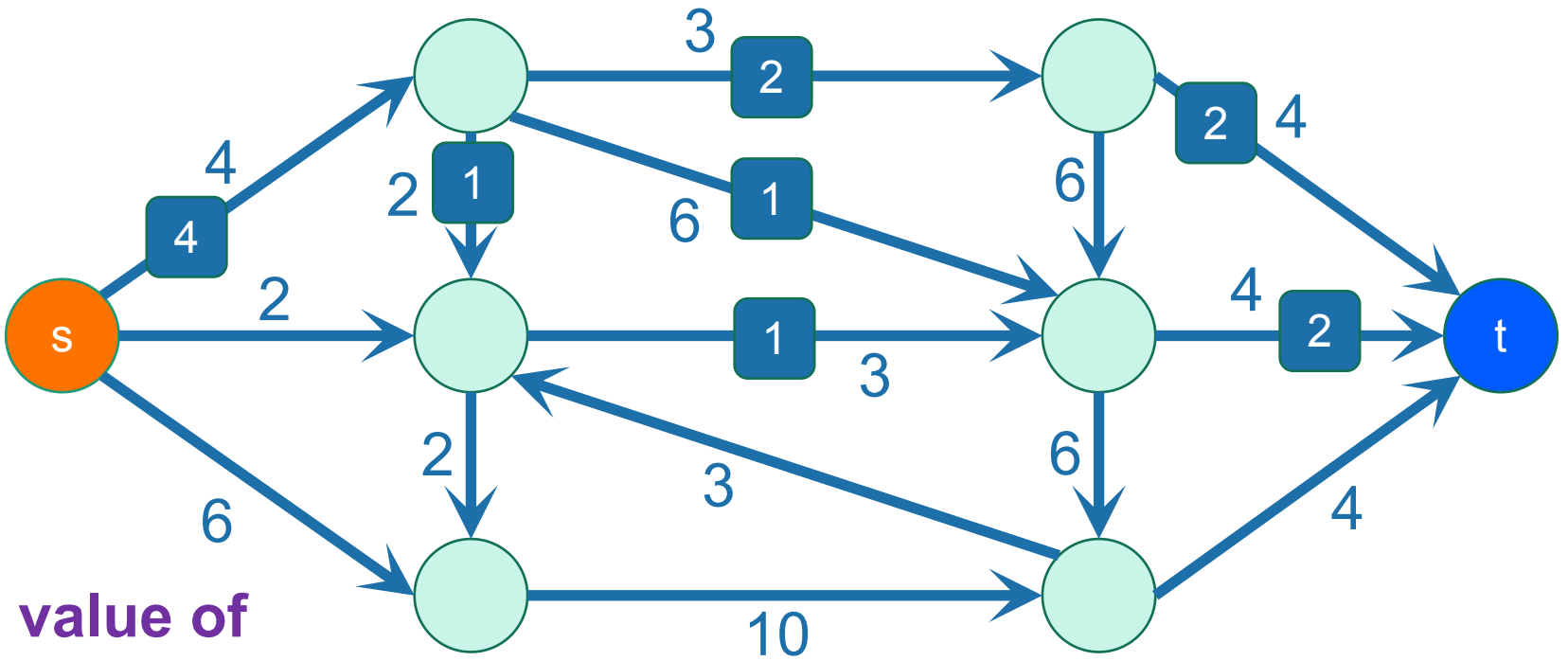


Flows

- 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,

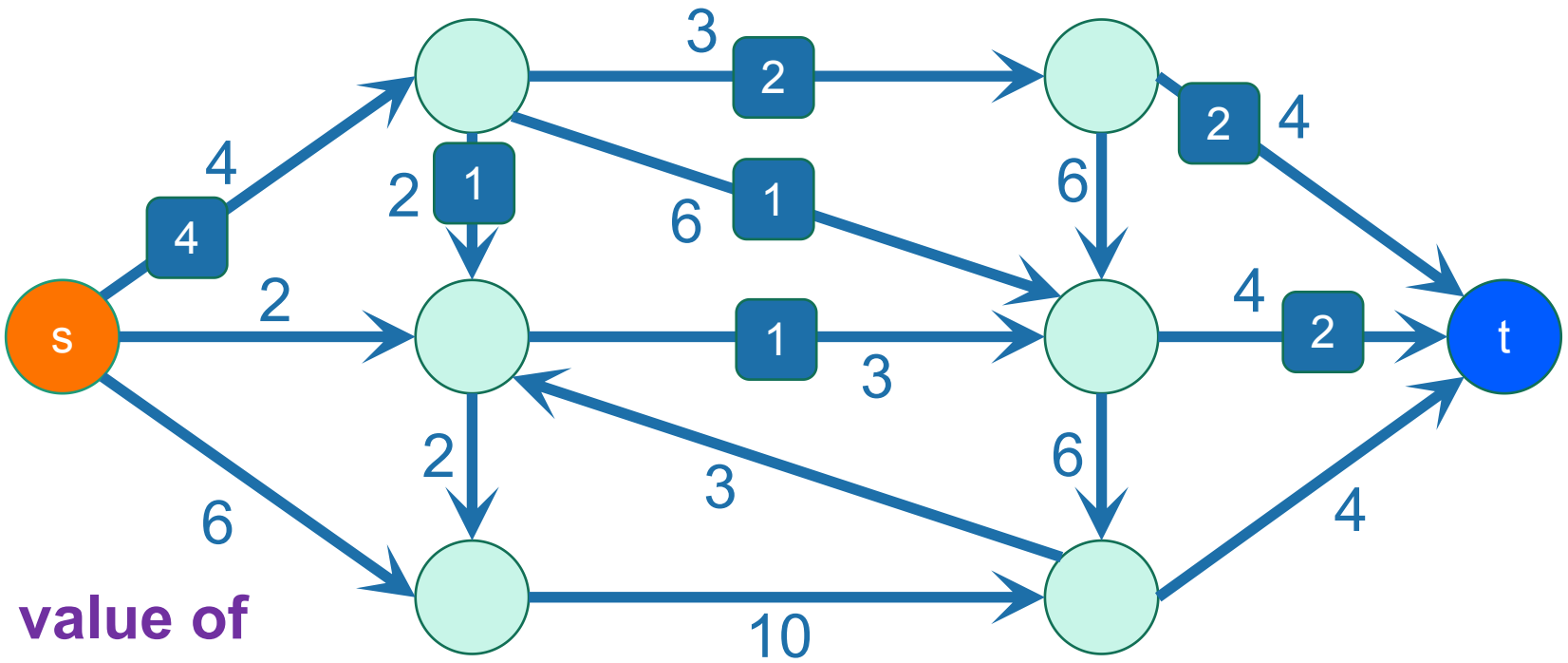
stuff you put in
=
stuff you take out.



The value of this flow is 4.

A maximum flow
is a flow of maximum value.

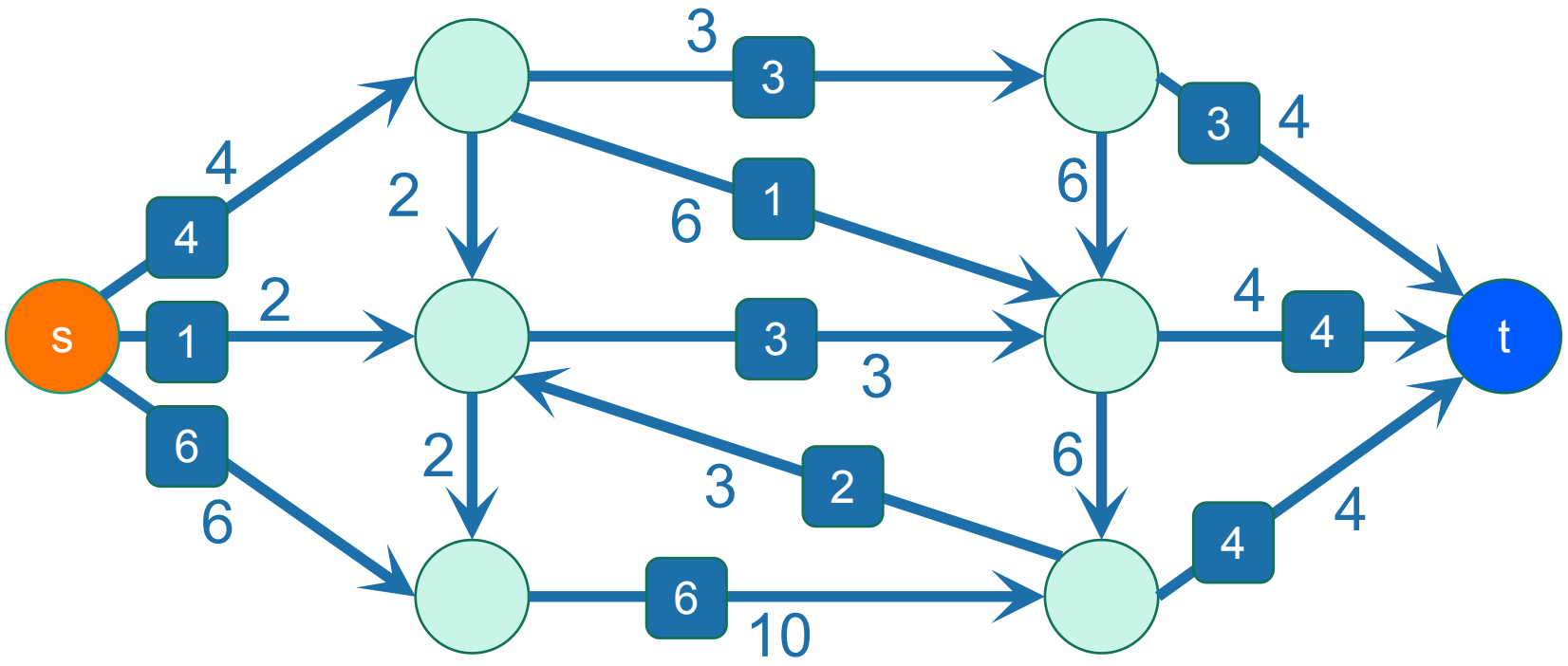
- This example flow is pretty wasteful, I'm not utilizing the capacities very well.



The value of this flow is 4.

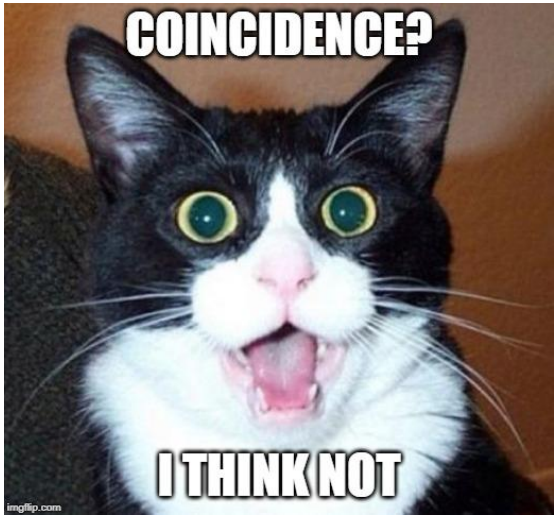
A maximum flow
is a flow of maximum value.

- This one is maximal.
  - It has value 11.

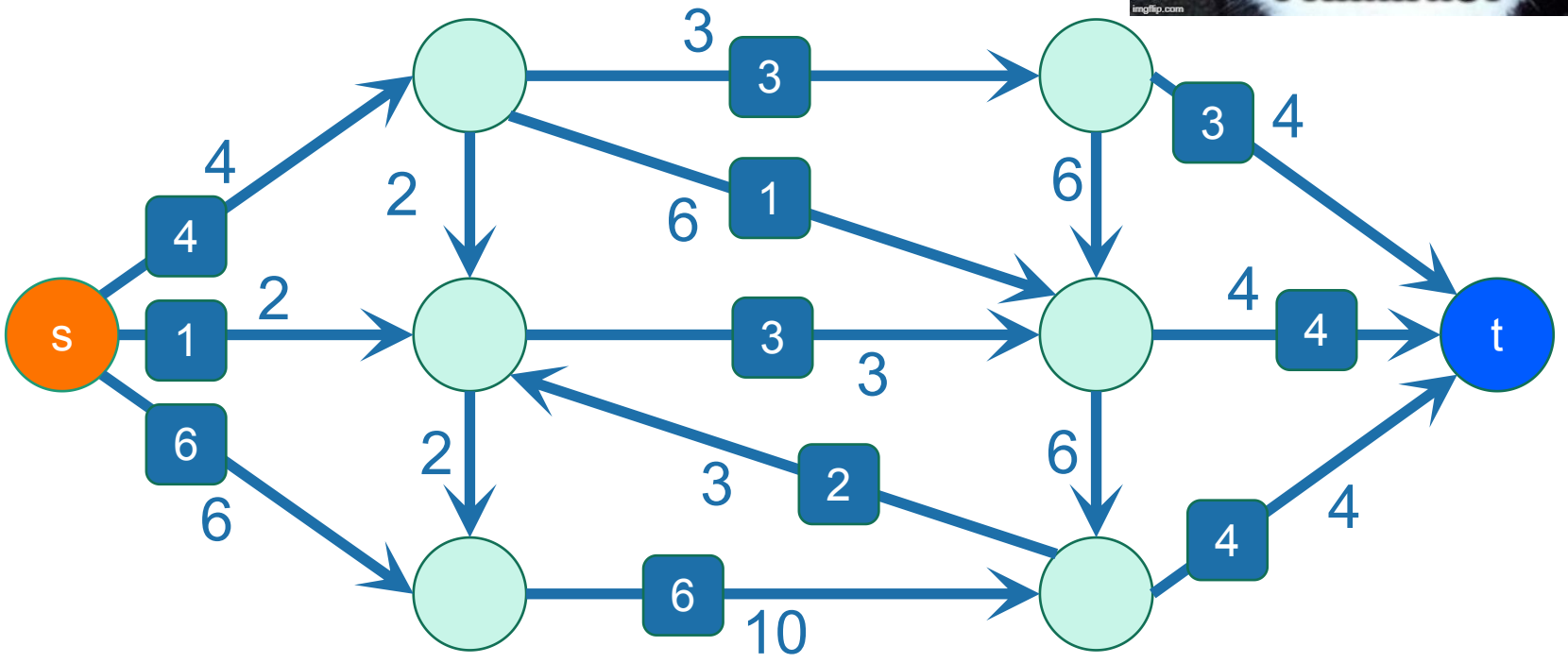


A maximum flow
is a flow of maximum value.

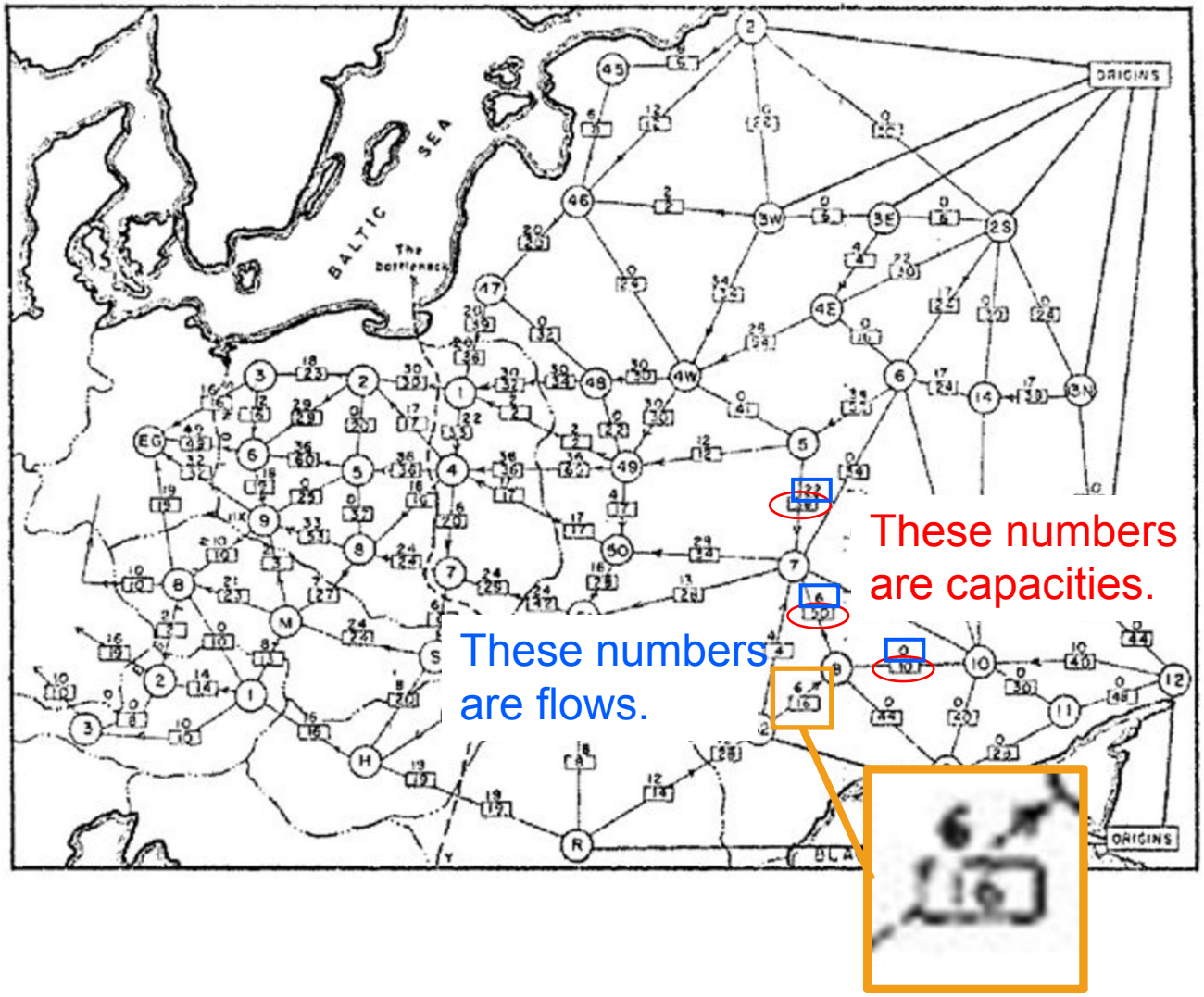
That's the same as the minimum cut in this graph!



- This one is maximal.
  - It has value 11.



Example



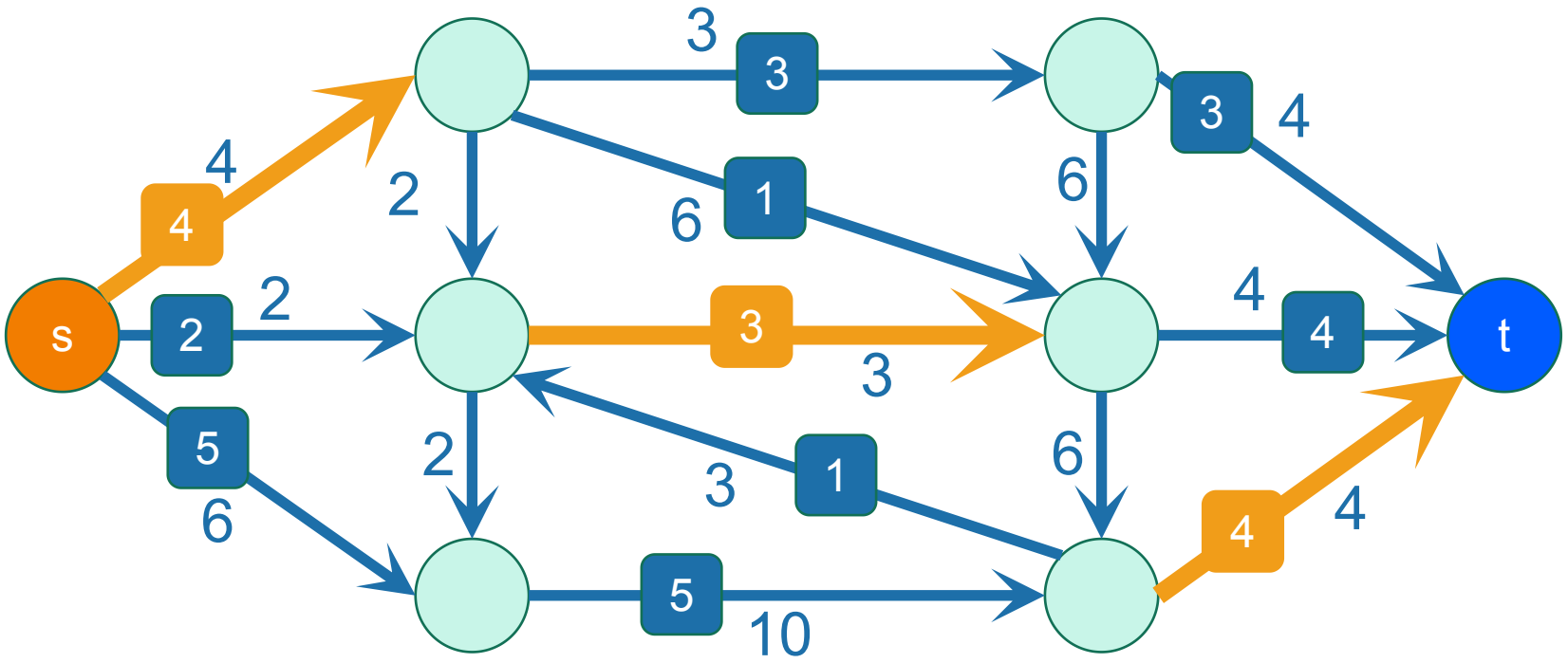
- 1955 map of rail networks from the Soviet Union to Eastern Europe.
  - Declassified in 1999.
  - 44 vertices, 105 edges
- The Soviet Union wants to route supplies from suppliers in Russia to Eastern Europe as efficiently as possible

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.

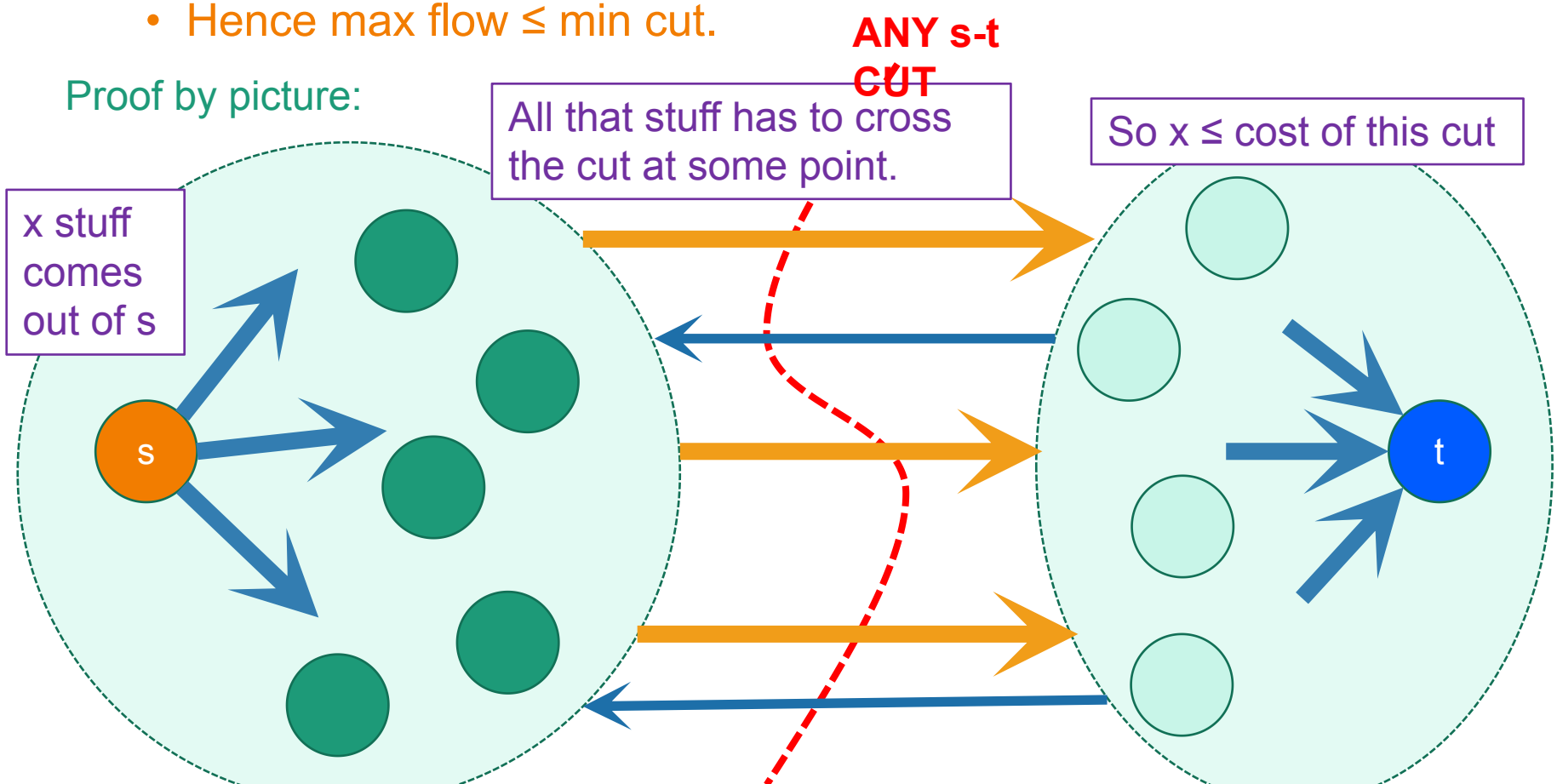
Intuition: in a max flow, the min cut better fill up, and this is the bottleneck.



Proof of Min-Cut Max-Flow Thm

- **Lemma1:**
  - 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

- **Lemma1:**
  - 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.

  - That was proof-by-picture.
  - See the notes for the proof-by-proof.
    - You are **not** responsible for the proof-by-proof on the final.

Proof of Min-Cut Max-Flow Thm

- **Lemma1:**
  - 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.

- **The theorem is stronger:**
  - max flow = min cut
  - Proof by algorithm
    - up next!

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.

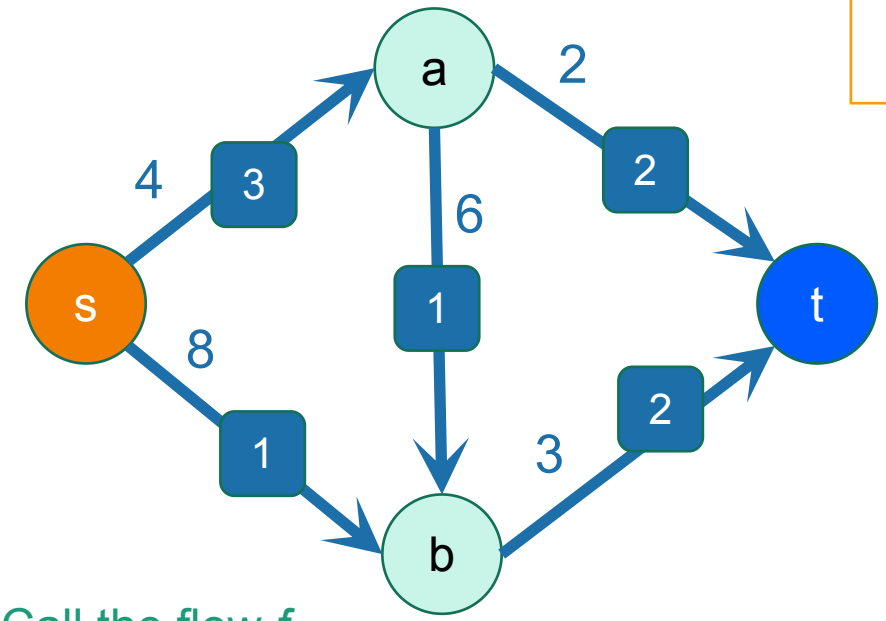
Assume for today that we don't have edges like this, although it's not necessary.

Tool: Residual networks

Say we have a flow

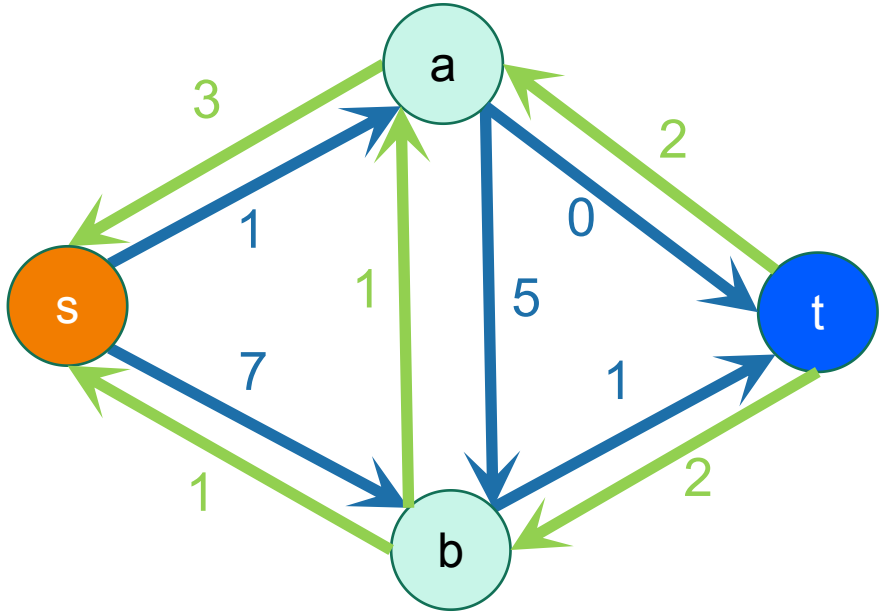
$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & else \end{cases}$$

- $f(u,v)$ 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*

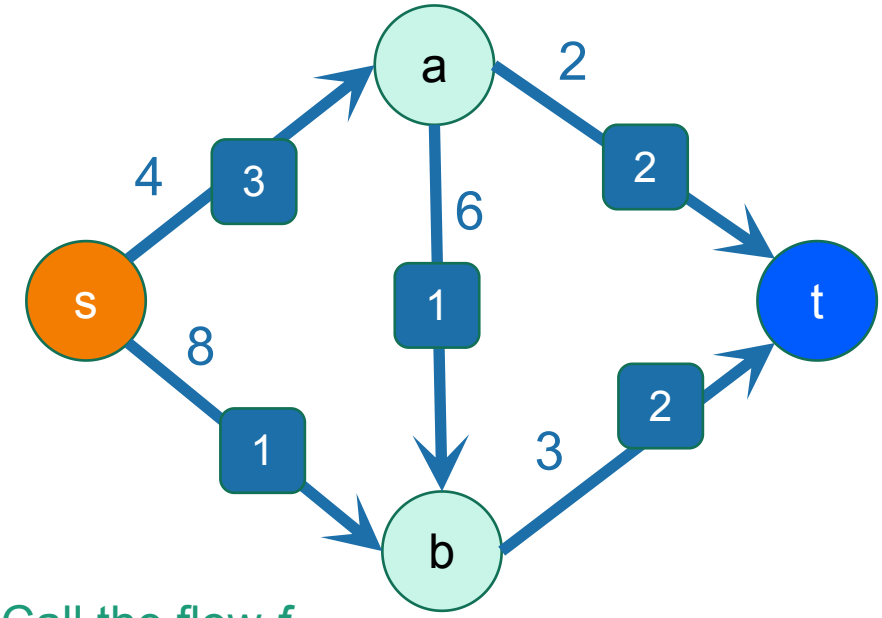
Create a new **residual network** from this flow:



Call the graph G_f

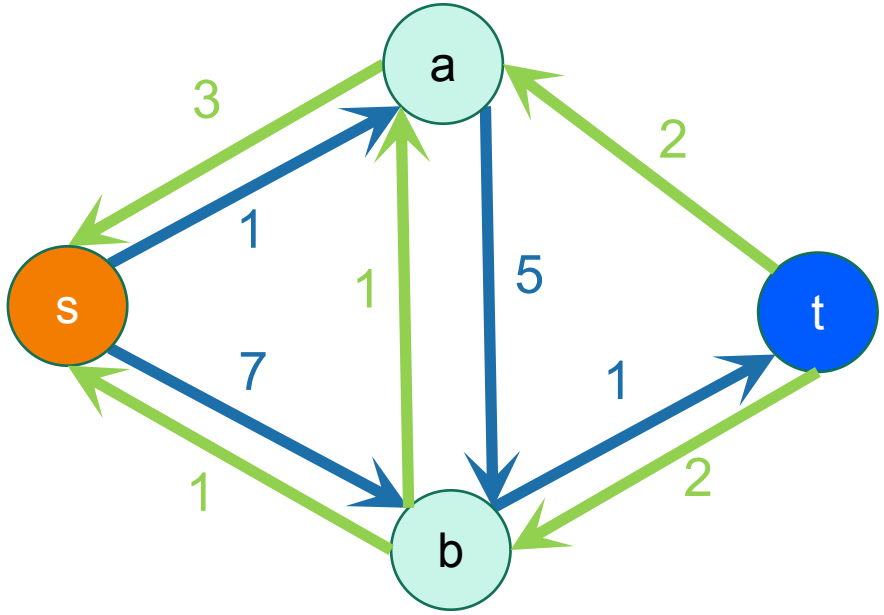
Tool: Residual networks

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.



Call the graph G_f

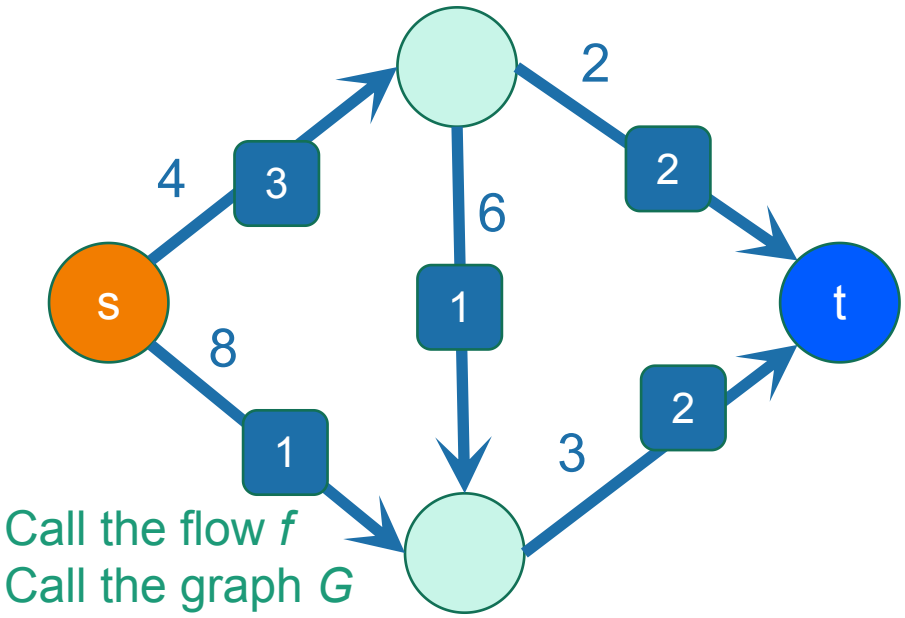
Create a new **residual network** from this flow:

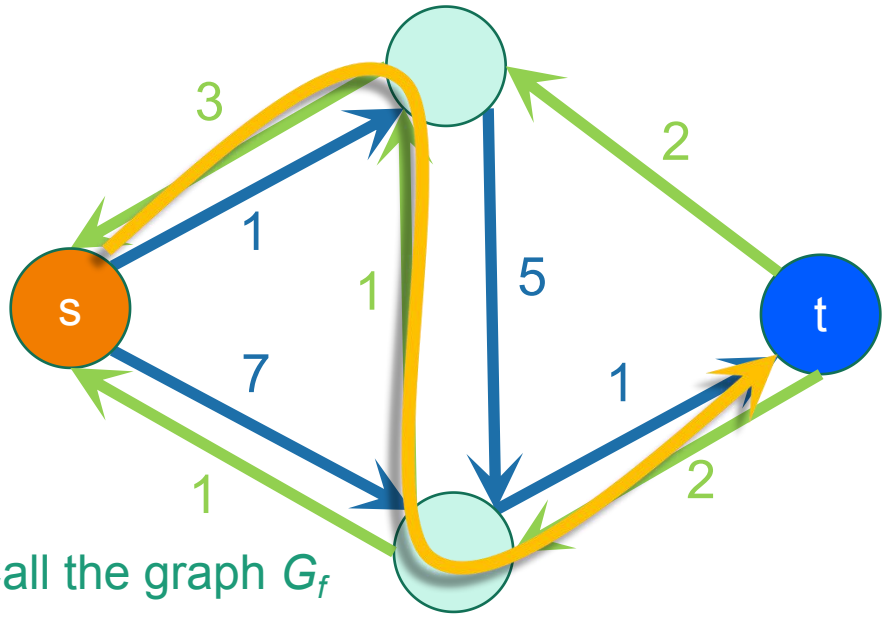
Why do we care about residual networks?

Lemma:

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

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



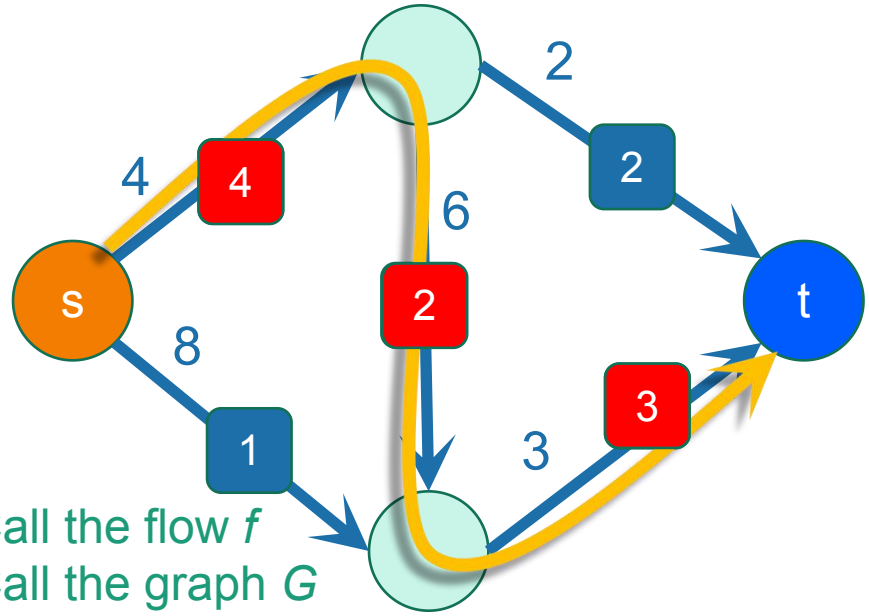


Why do we care about residual networks?

Lemma:

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

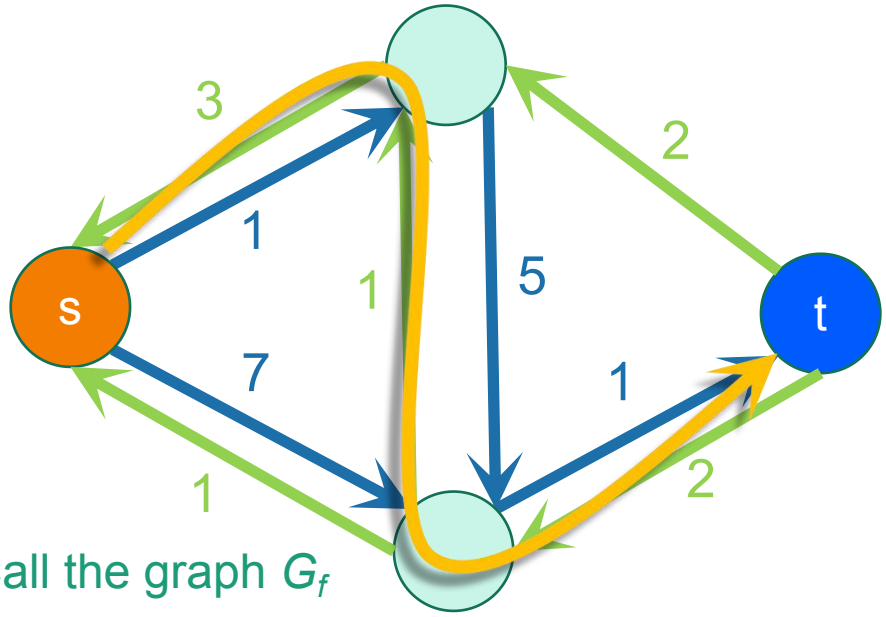
To see that this flow is not maximal, notice that we can improve it by sending one more unit of stuff along this path:



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 the graph G_f

Why do we care about residual networks?

Lemma:

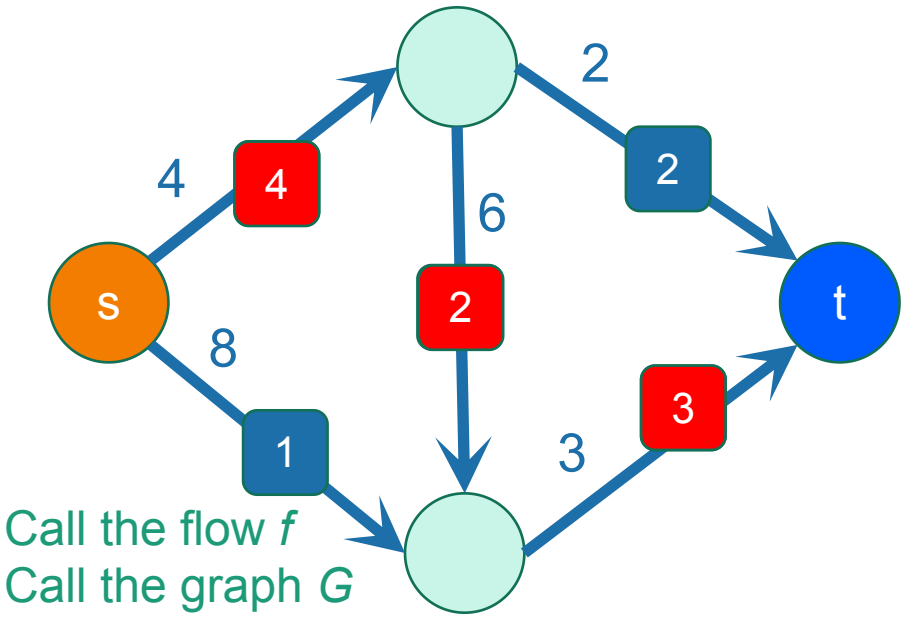
- t is not reachable from s in G_f f is a max flow.

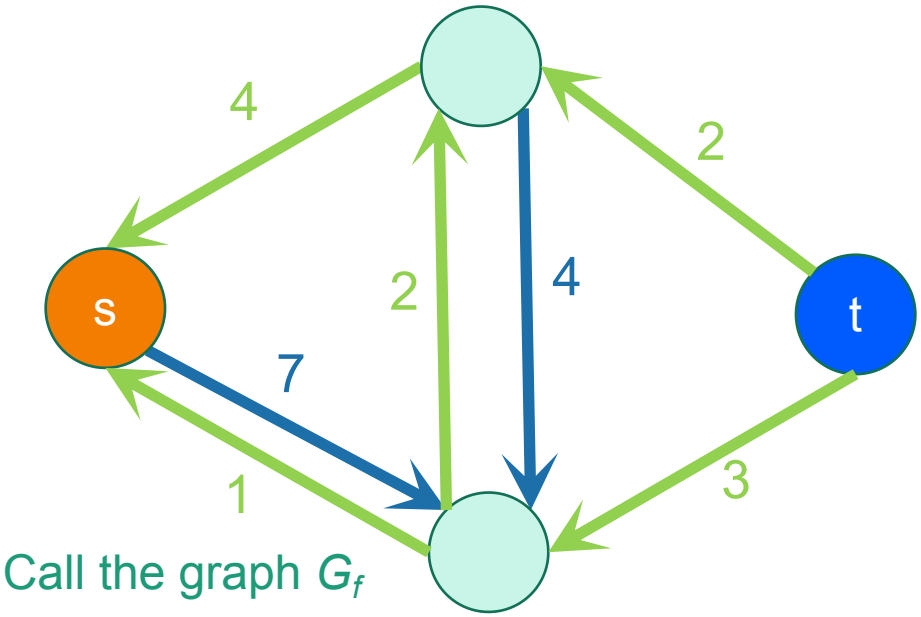
Example:

Now we get this residual graph:

Now we can't reach t from s.

So the lemma says that f is a max flow.





Let’s prove the Lemma

- t is not reachable from s in G_f **<->** *f* is a max flow.

Lemma:

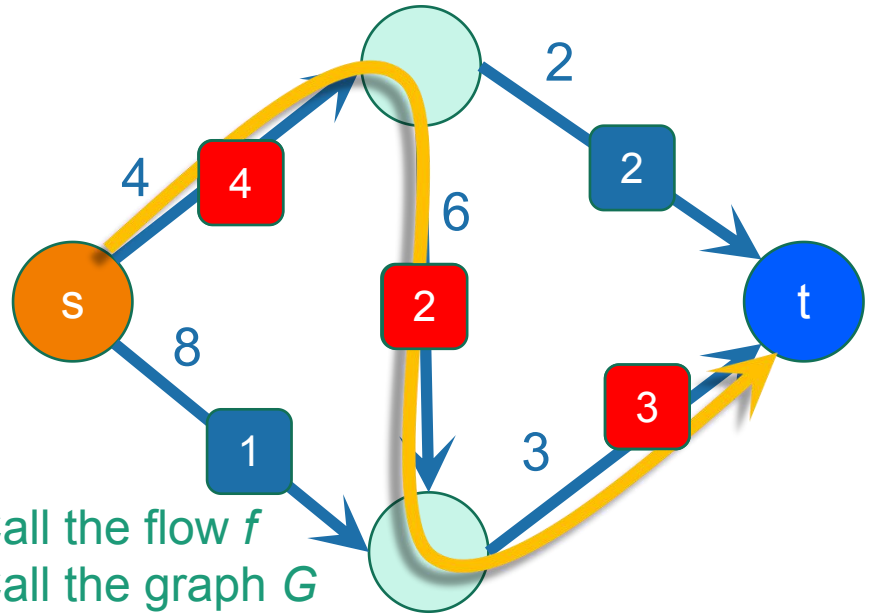
first this direction

We will prove the contrapositive

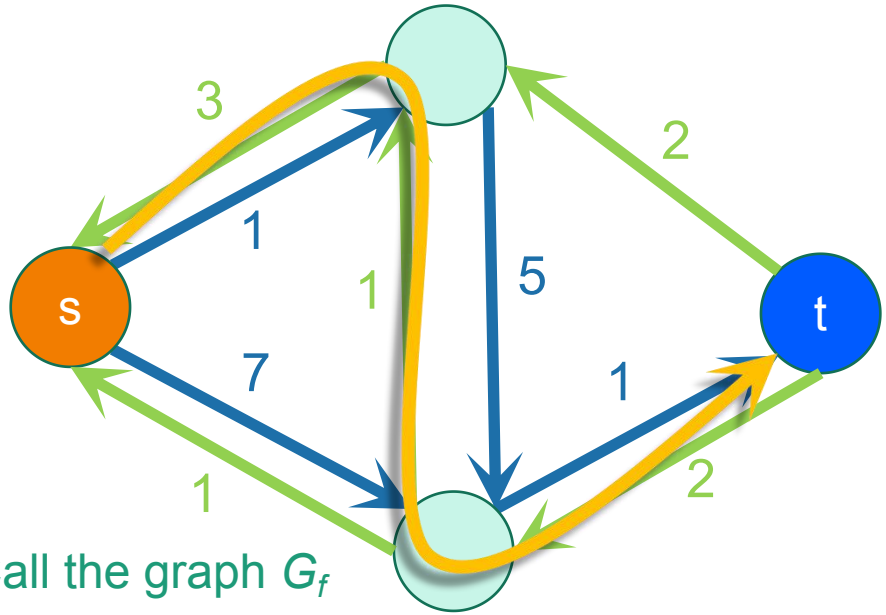
t is not reachable from s in G_f ***f*** is a max flow.

- Suppose there is a path from s to t in G_f.
  - This is called augmenting path.
- Claim: if there is an augmenting path, we can increase the flow along the path.
- So do that and update 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*

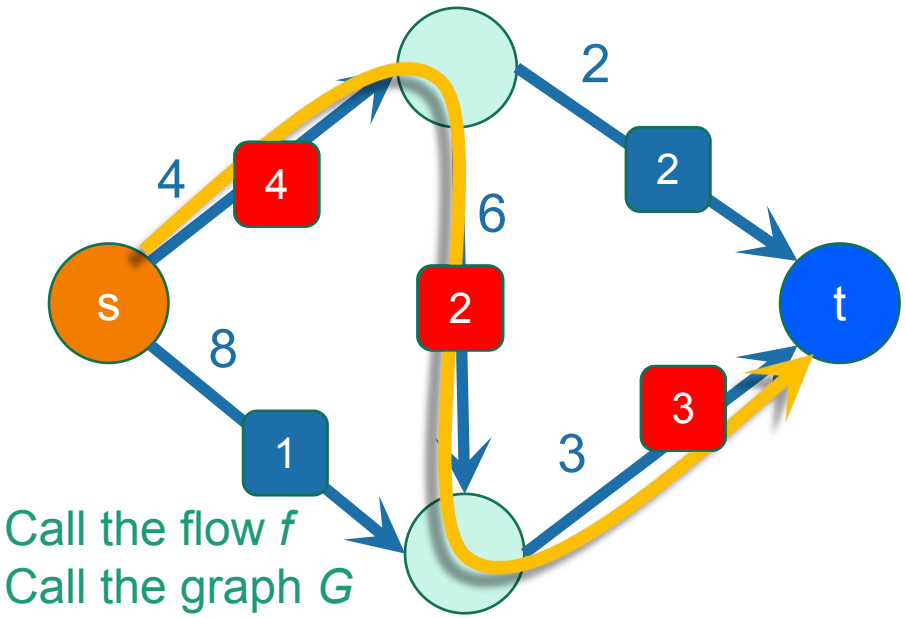


Call the graph G_f

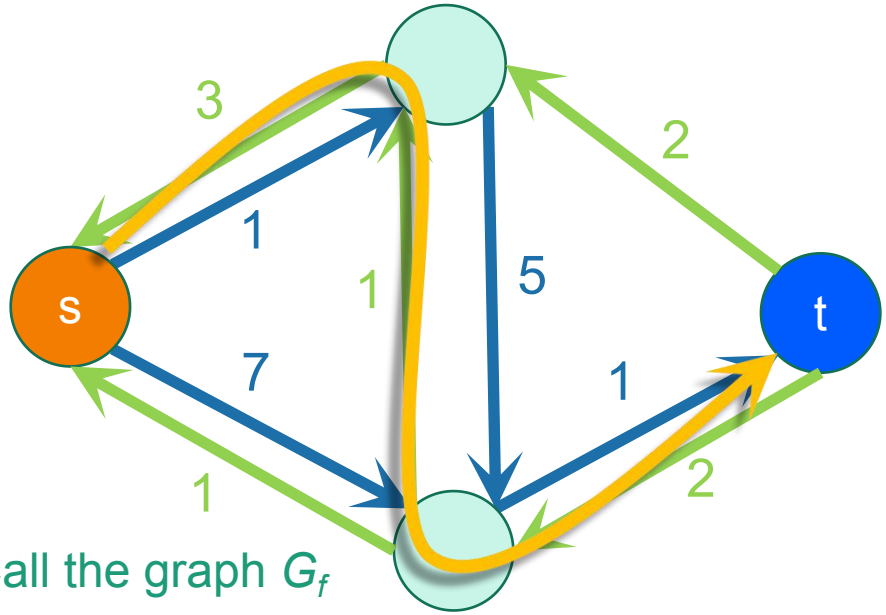
Claim:

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

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



Call the flow *f*
Call the graph *G*



Call the graph G_f

- Every edge on the path was below capacity, so just increase the flow on all the edges.

Claim:

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

Check that this always makes a bigger flow!

Ollie the over-achieving ostrich

- 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, $f'(u,v) \leftarrow f(u,v) - x.$
  - **return** *f'*

That proves the **claim**

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

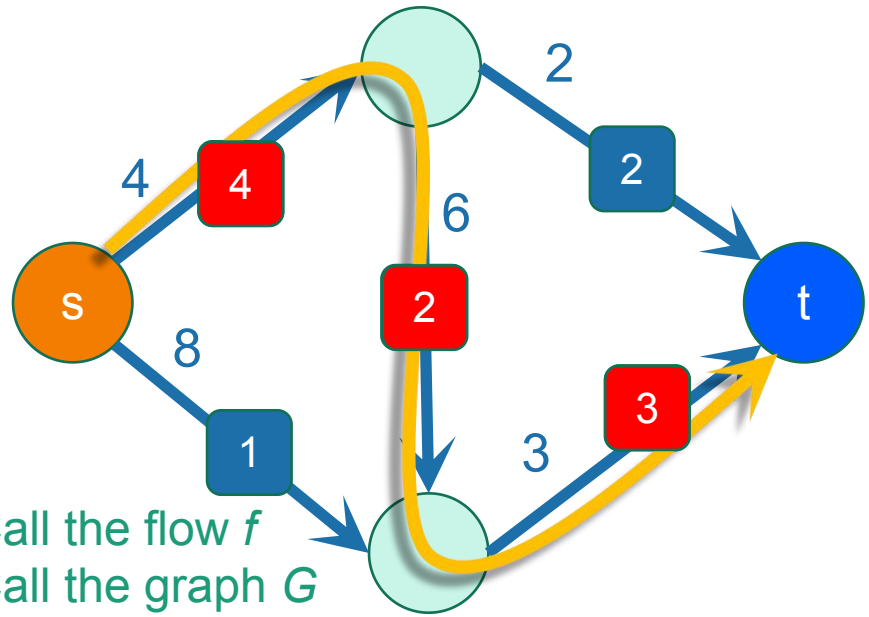
Lemma:

first this direction <-

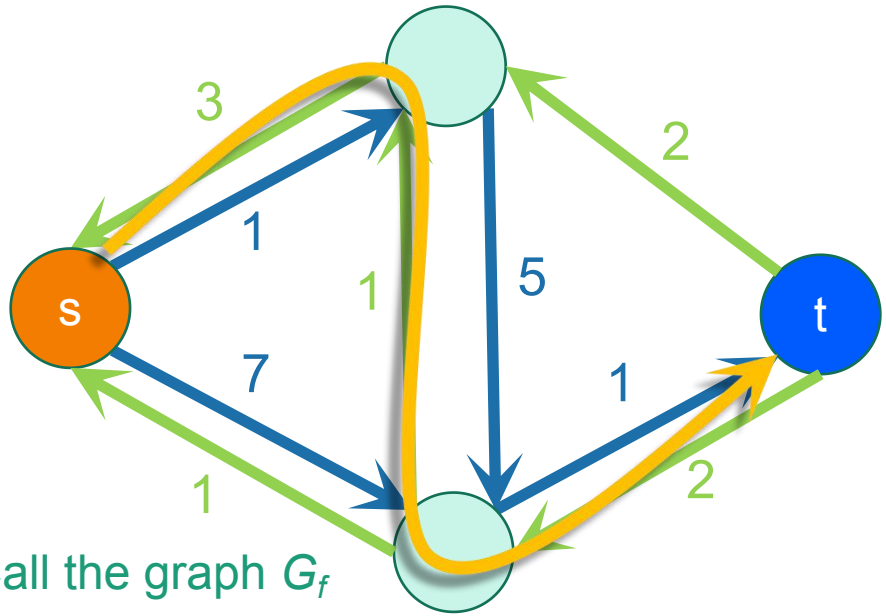
We will prove the contrapositive

t is not reachable from s in G_f <- f is a max flow.

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



Call the flow f
Call the graph G



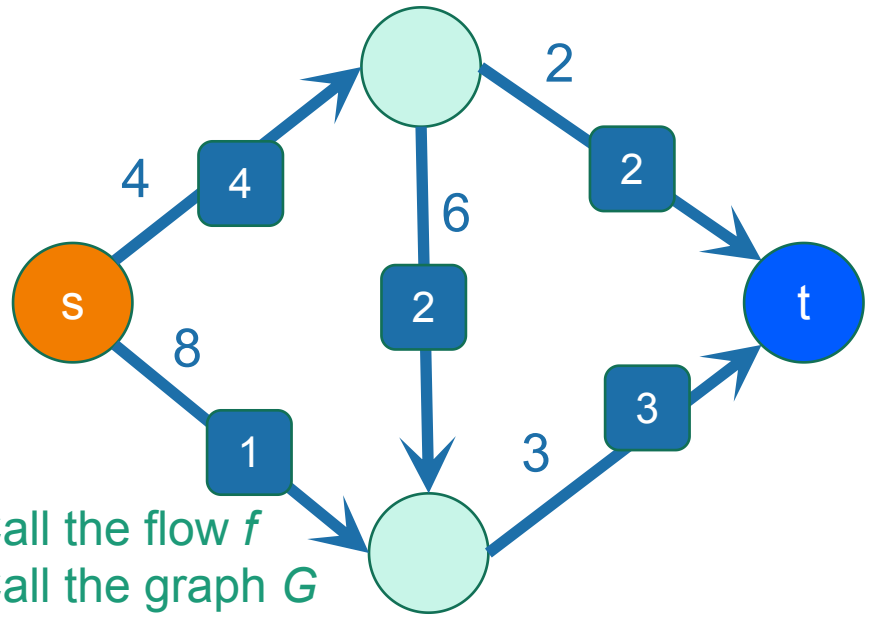
Call the graph G_f

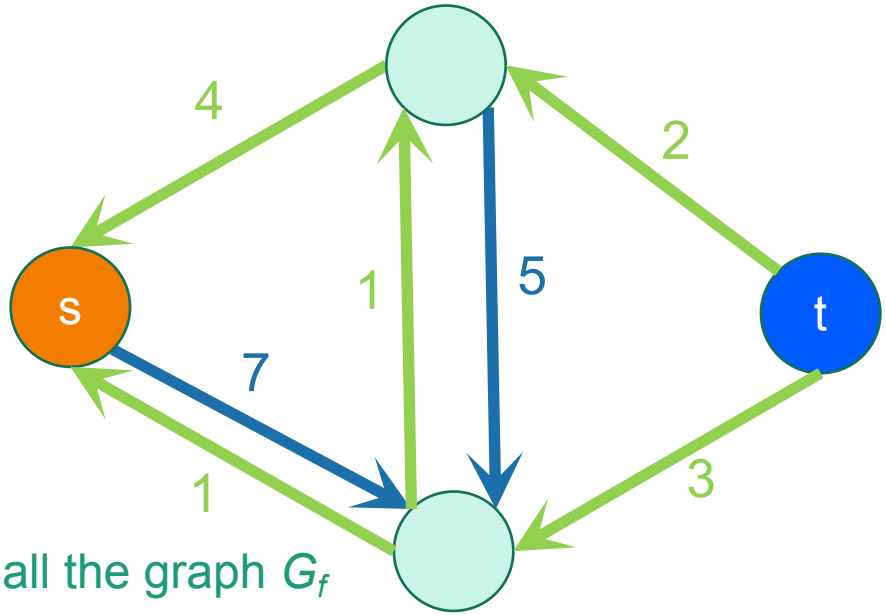
Lemma: now this direction ->

t is not reachable from s in G_f -> ***f*** is a max flow.

- Suppose there is no path from s to t in G_f.
- Consider the cut given by:

  {things reachable from s} , **{things not reachable from s}**





Lemma:

now this direction ->

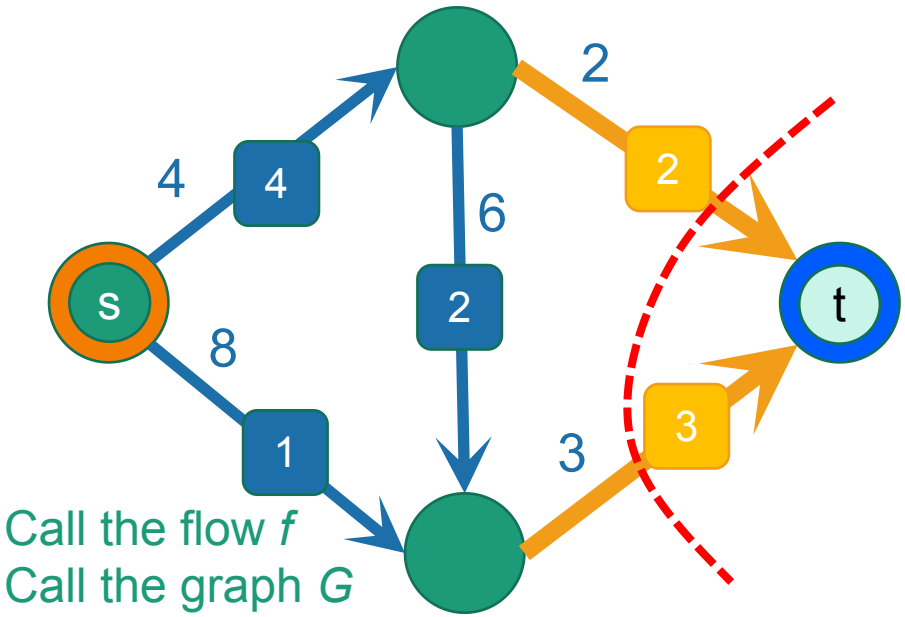
t is not reachable from s in G_f -> *f* is a max flow.

- Suppose there is no path from s to t in G_f.
- Consider the cut given by:

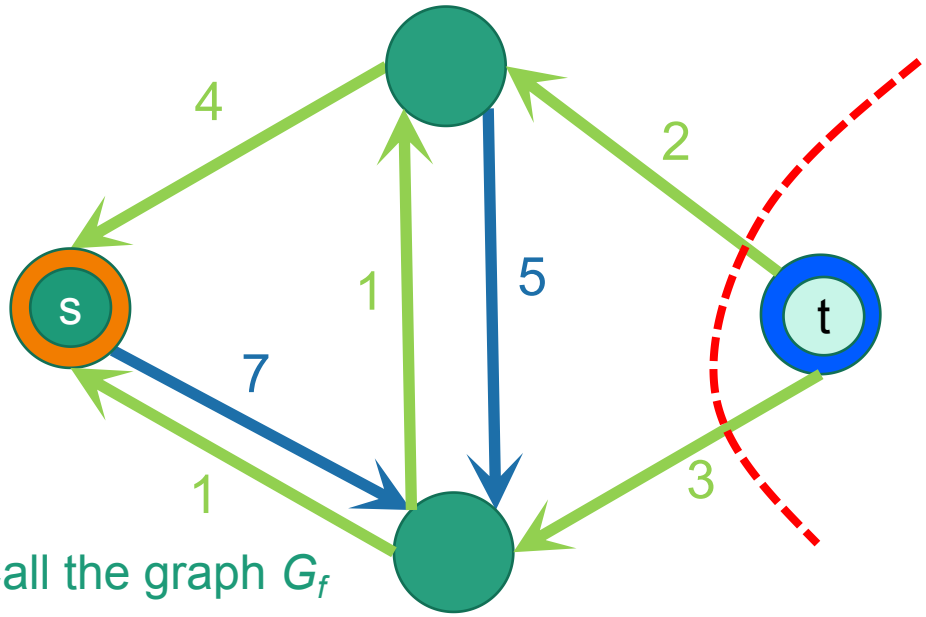
  t lives here

  {things reachable from s} , **{things not reachable from s}**
- The flow from s to t is **equal** to the cost of this cut
  - Similar to proof-by-picture we saw before:
    - All the stuff has to cross the cut
    - The edges in the cut are full because they don't exist in G_f
- **thus**: this flow value = cost of this cut = cost of min cut = max flow

Lemma 1



Call the flow *f*
Call the graph *G*



Call the graph G_f

Lemma: now this direction ->

t is not reachable from s in G_f -> *f* is a max flow.

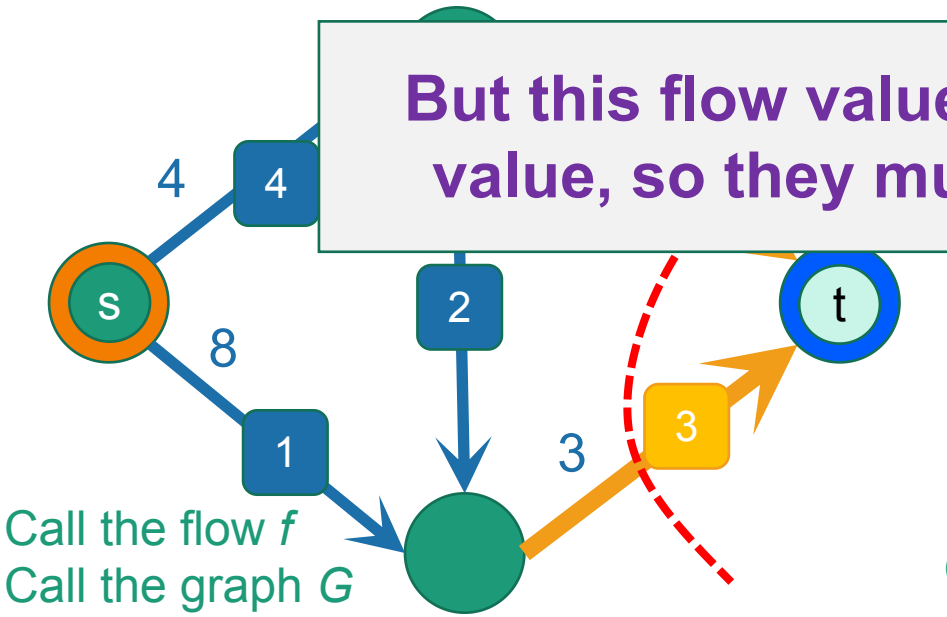
- Suppose there is no path from s to t in G_f.
- Consider the cut given by:

  t lives here

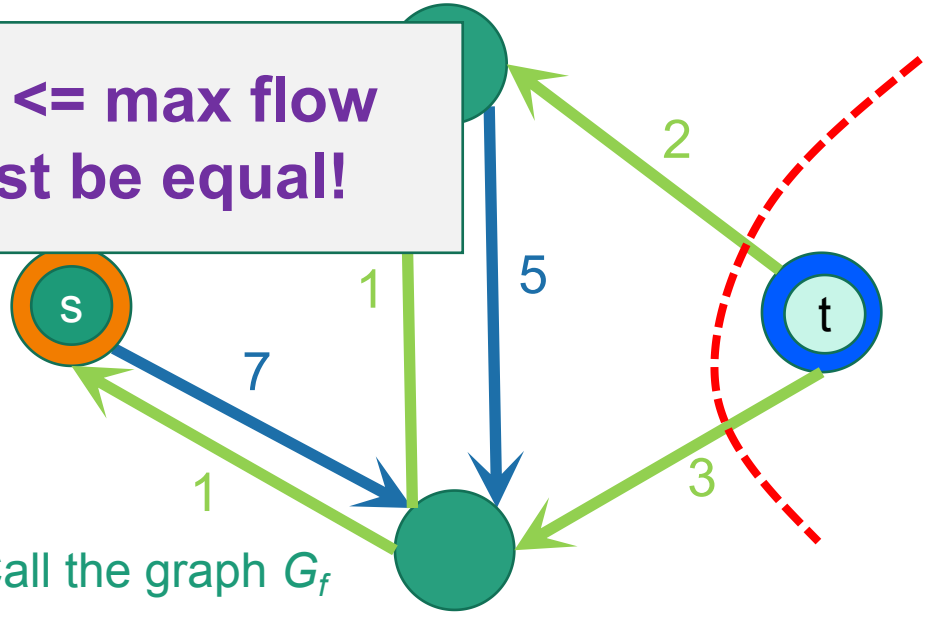
  {things reachable from s} , **{things not reachable from s}**
- The flow from s to t is **equal** to the cost of this cut
  - Similar to proof-by-picture we saw before:
    - All the stuff has to cross the cut
    - The edges in the cut are full because they don't exist in G_f
- **thus**: this flow value = cost of this cut = cost of min cut = max flow

Lemma 1

But this flow value <= max flow value, so they must be equal!



Call the flow *f*
Call the graph *G*



Call the graph G_f

We've proved:

- t is not reachable from s in G_f **<->** f is a max flow

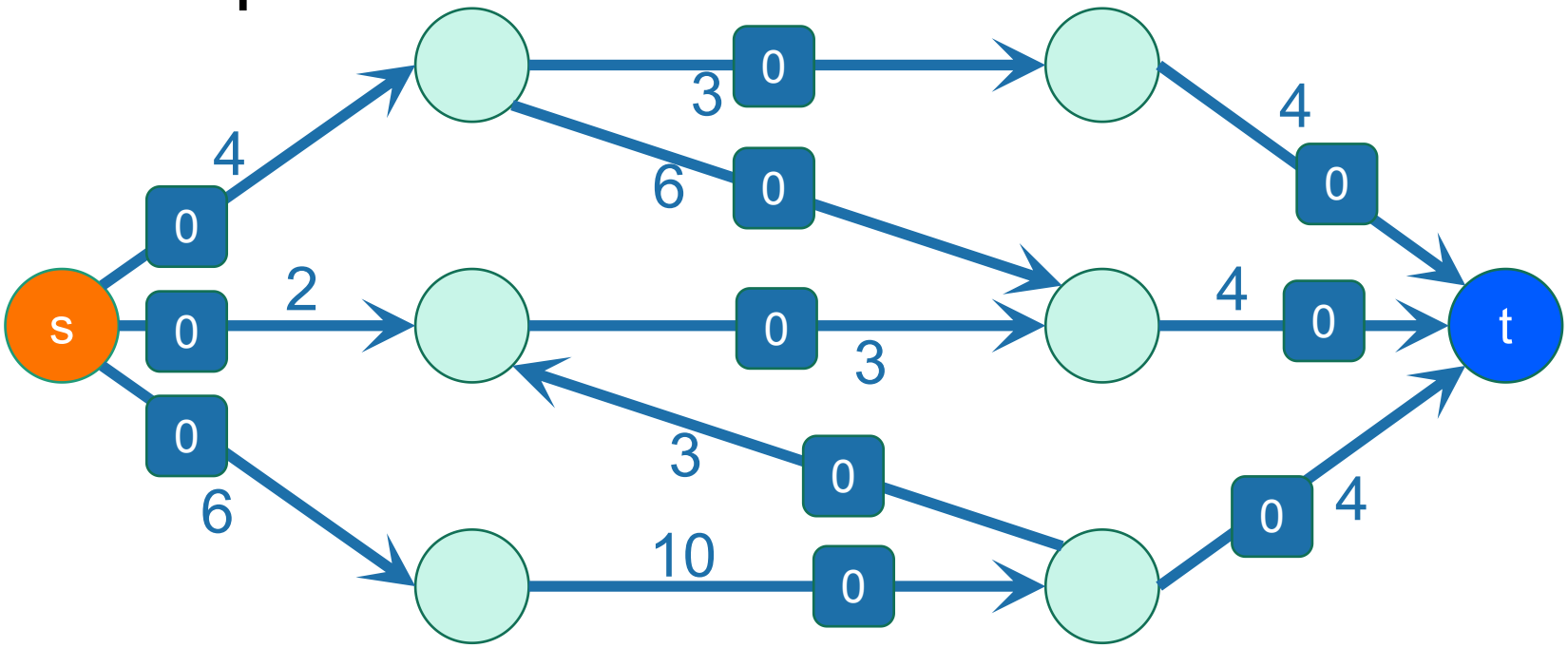
- This inspires an **algorithm**:
- Ford-Fulkerson(G):
  - f <-all zero flow.
  - G_f **<-** G
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f // eg, use BFS
    - f **<- increaseFlow**(P, f)
    - update G_f
  - return f

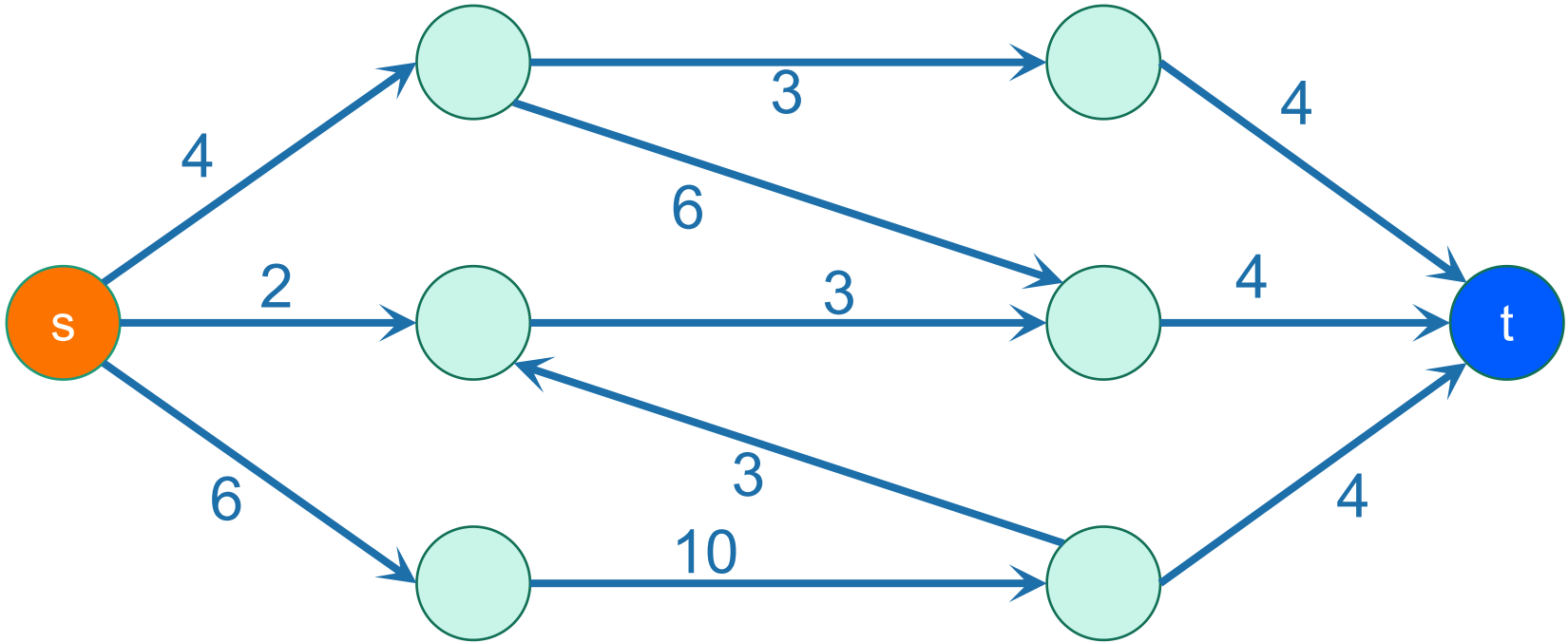
How do we choose which paths to use?

- 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
  - (Or may actually never terminate?)

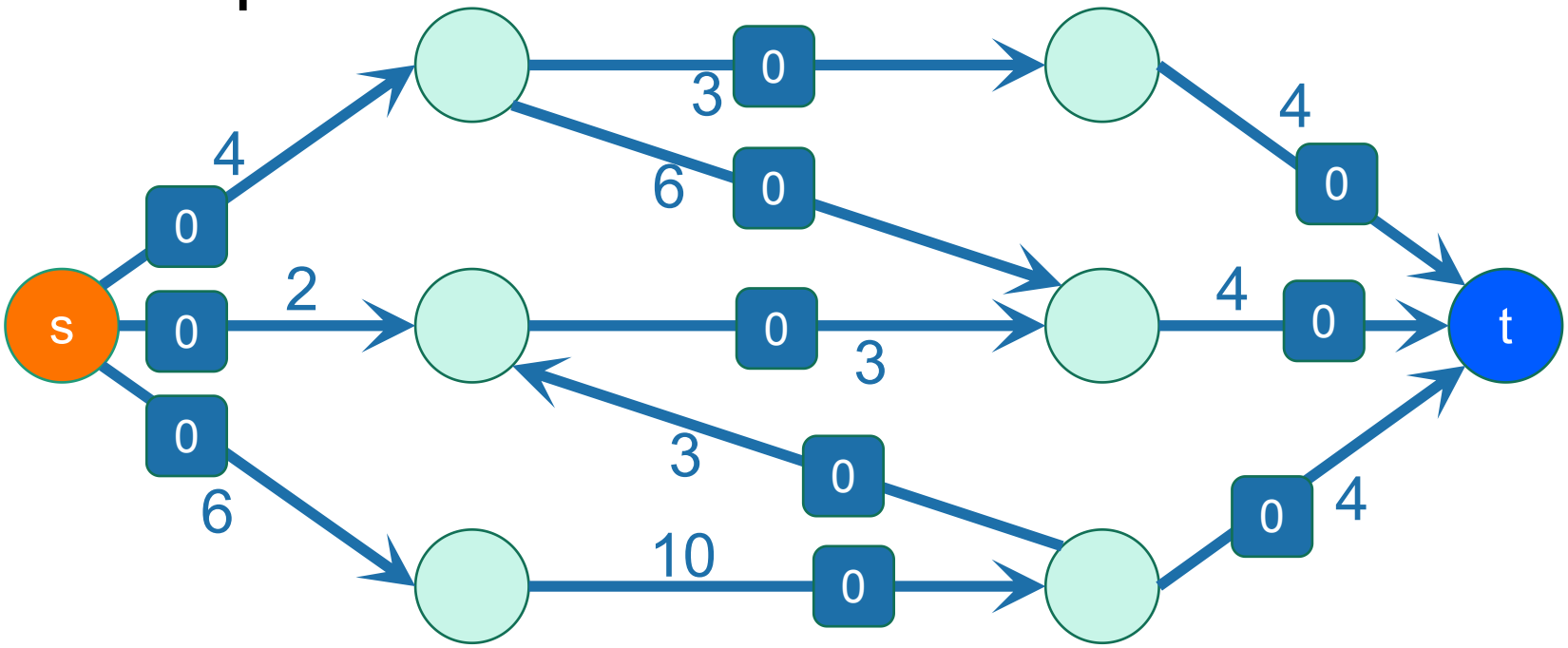
- We need to be careful with our path selection to make sure the algorithm terminates quickly.
  - Using BFS leads to the **Edmonds-Karp algorithm.** *(today)*
  - There's another way in the notes. *(optional)*

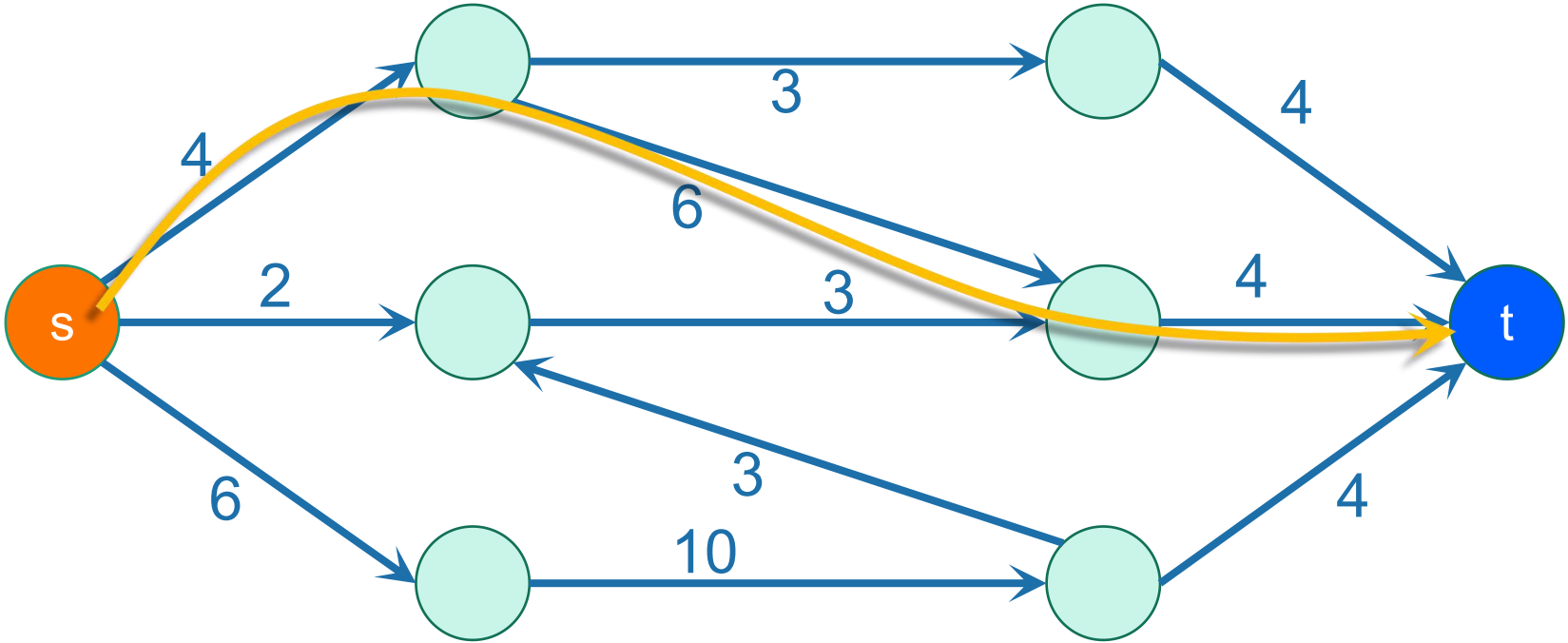
Example of Ford-Fulkerson



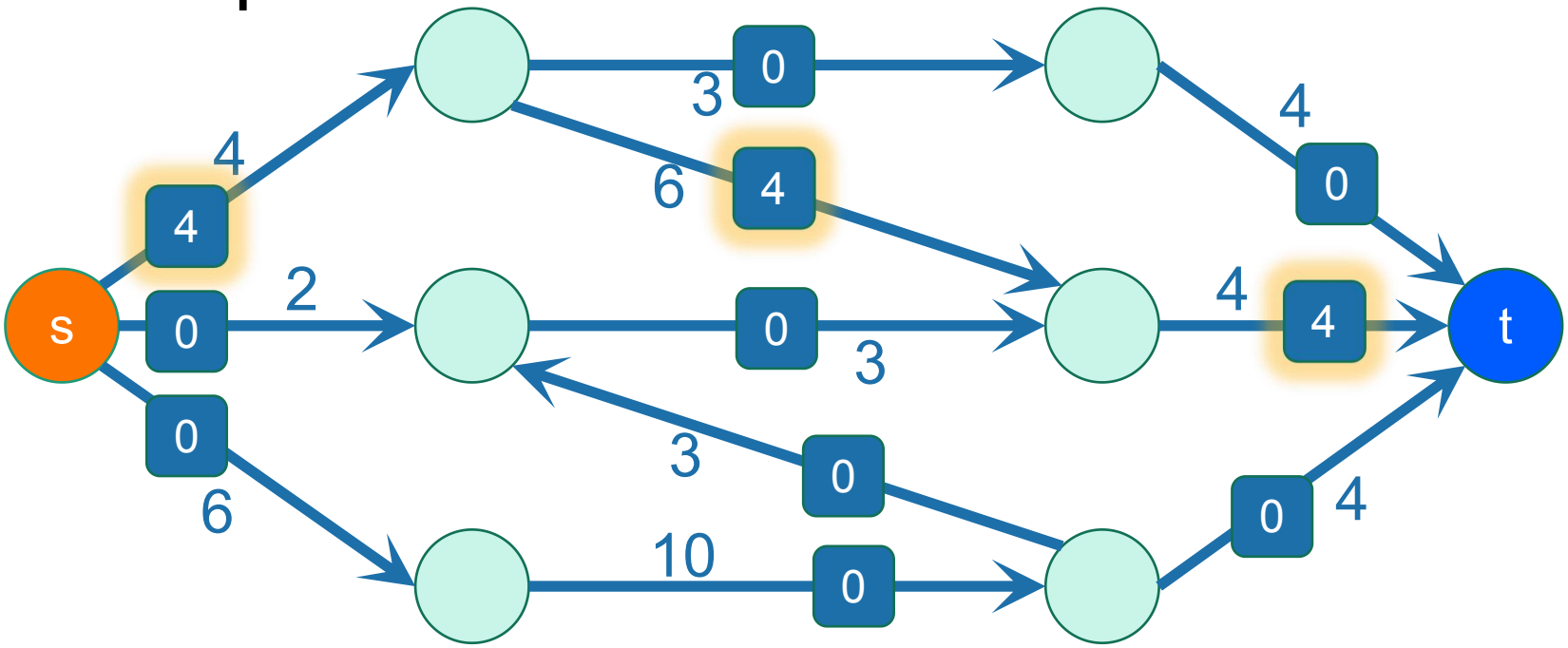


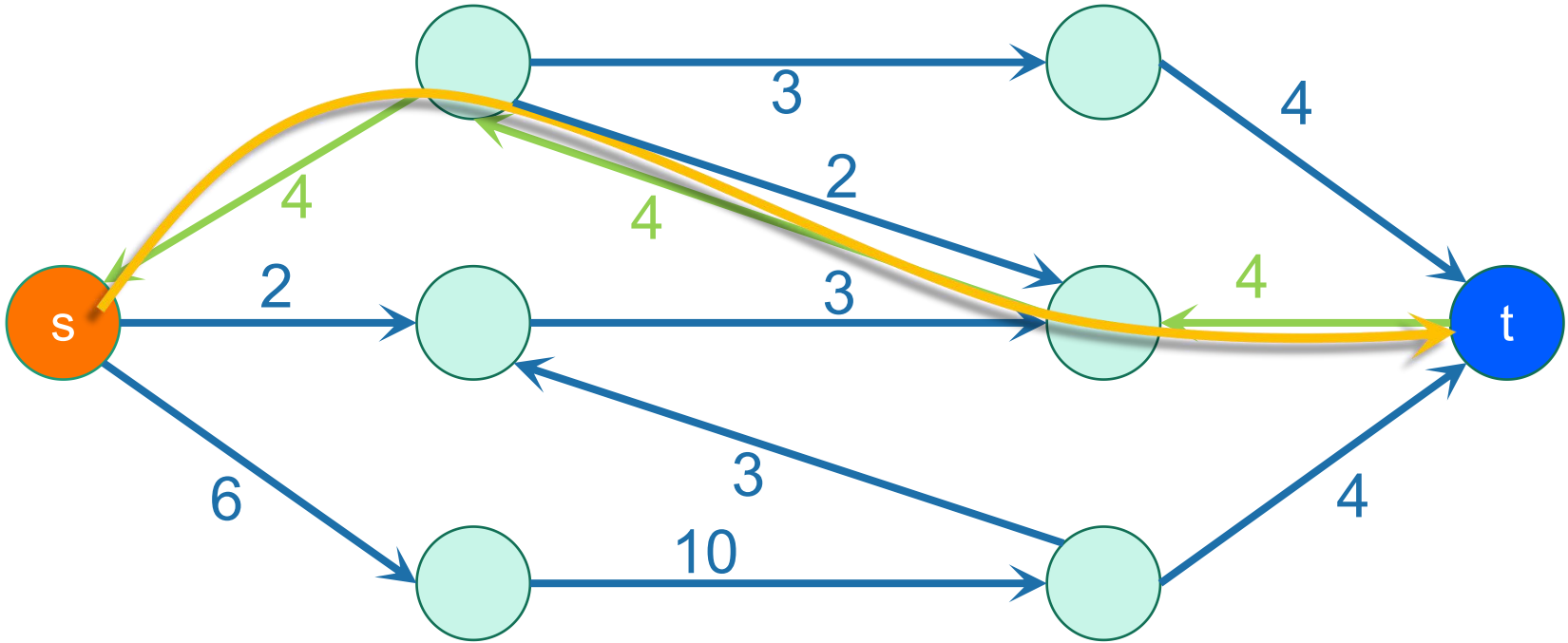
Example of Ford-Fulkerson



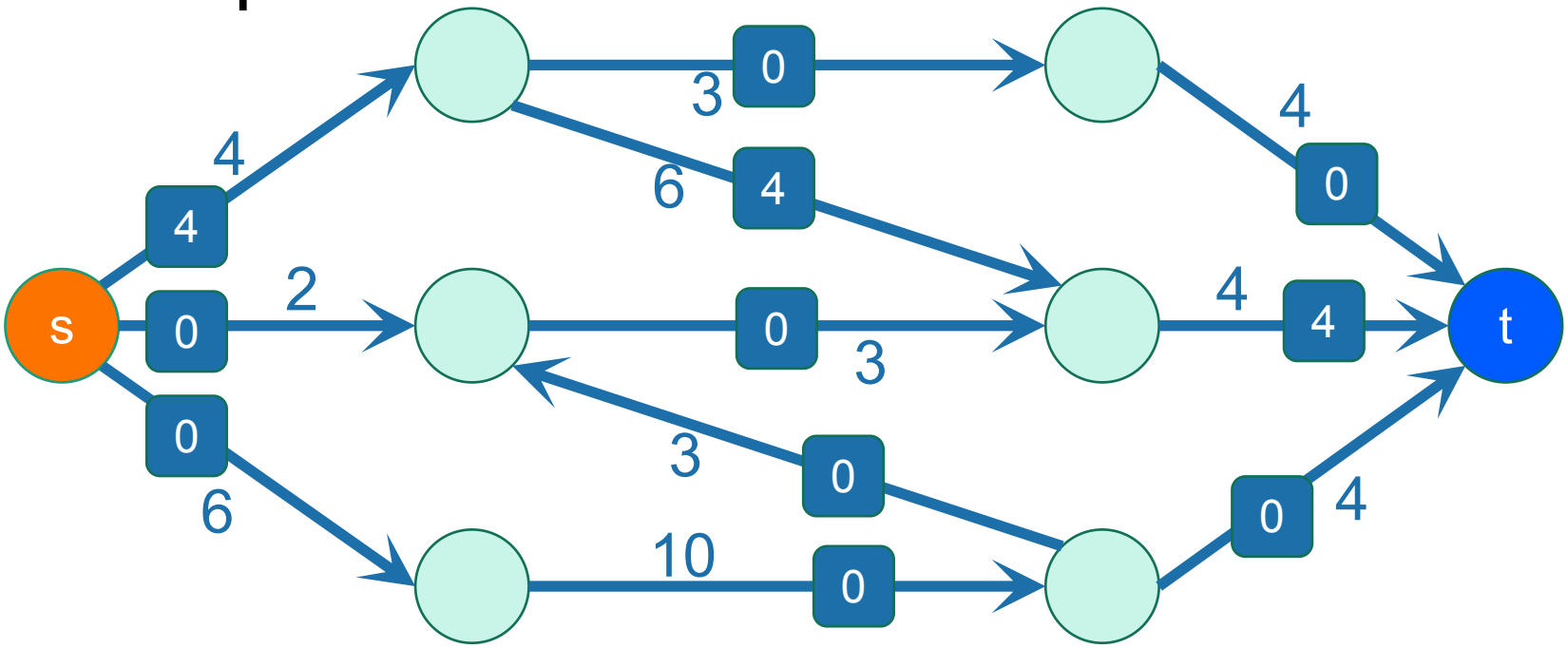


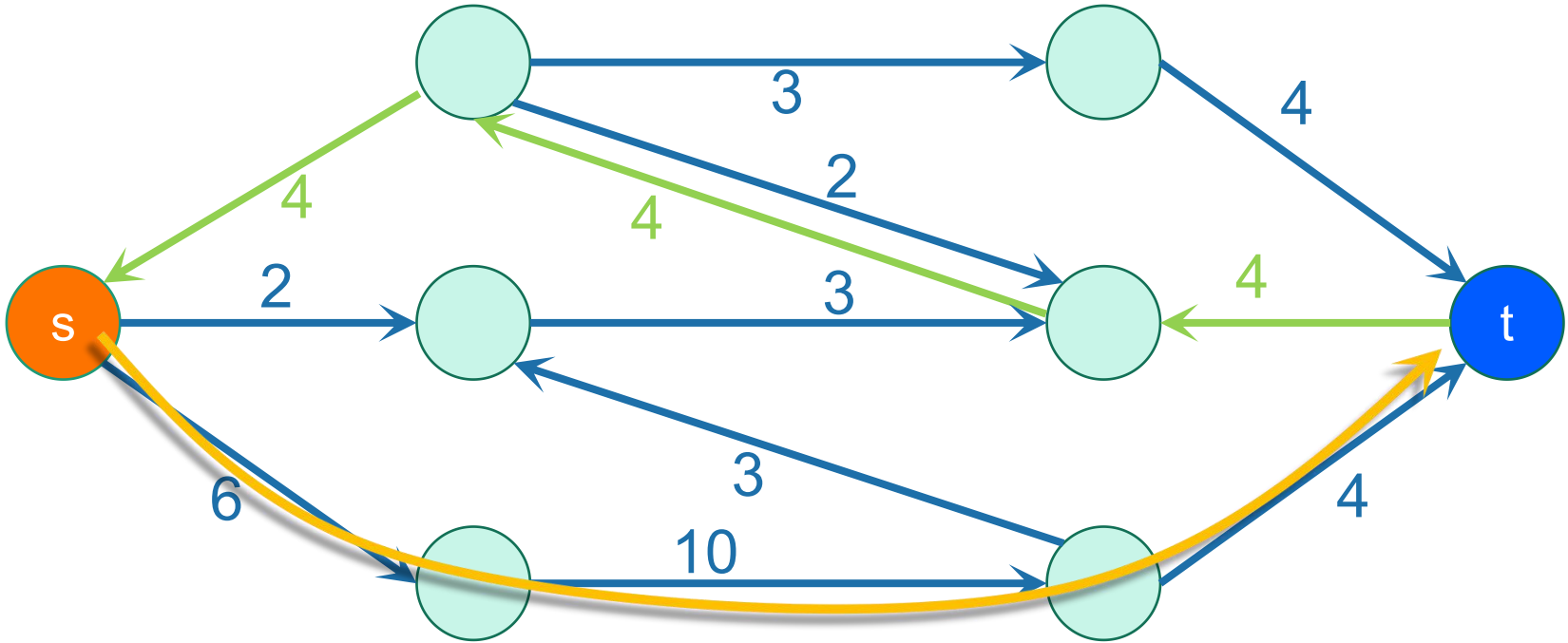
Example of Ford-Fulkerson



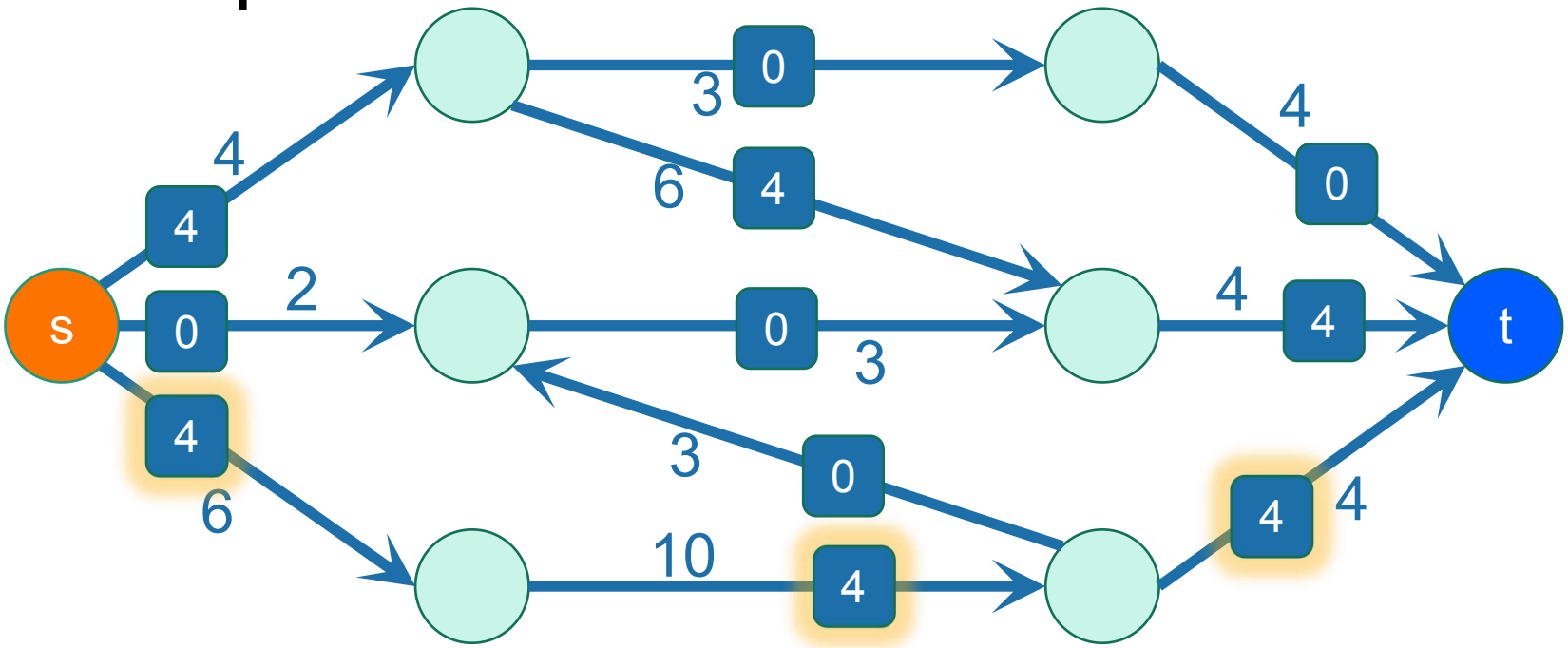


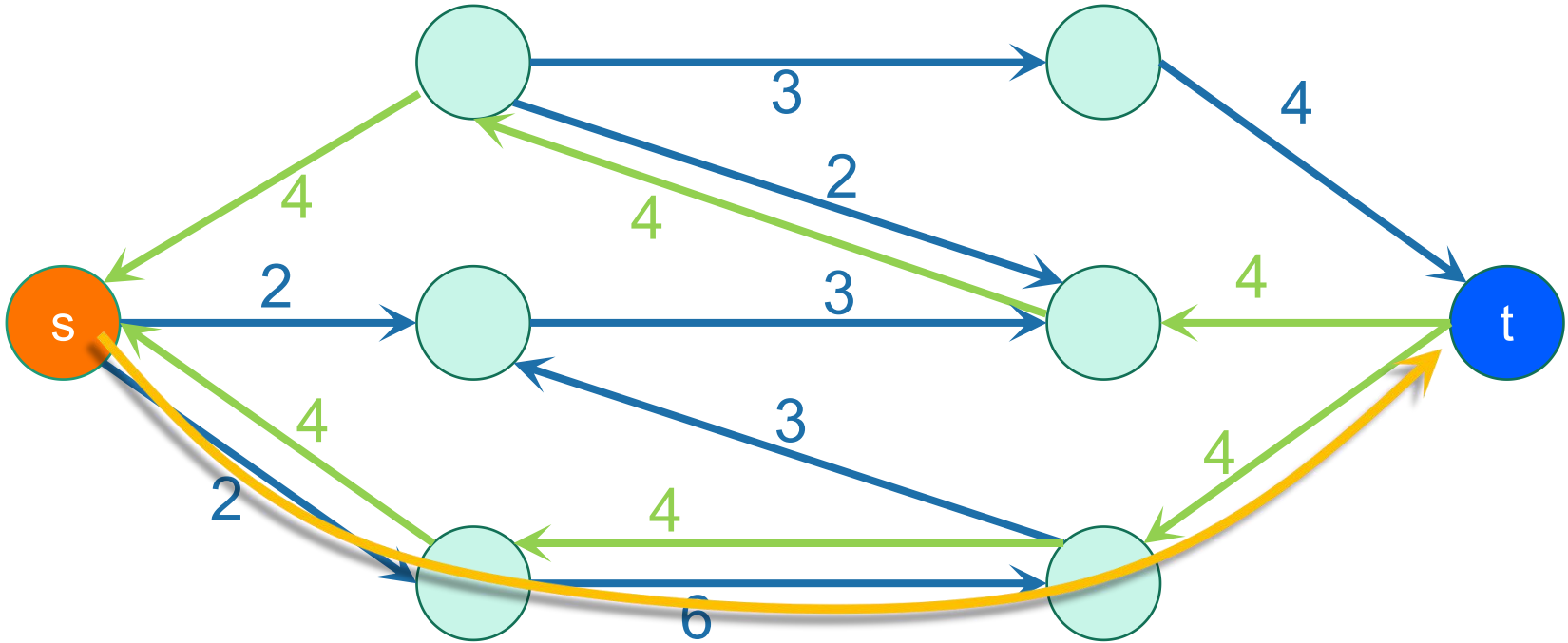
Example of Ford-Fulkerson



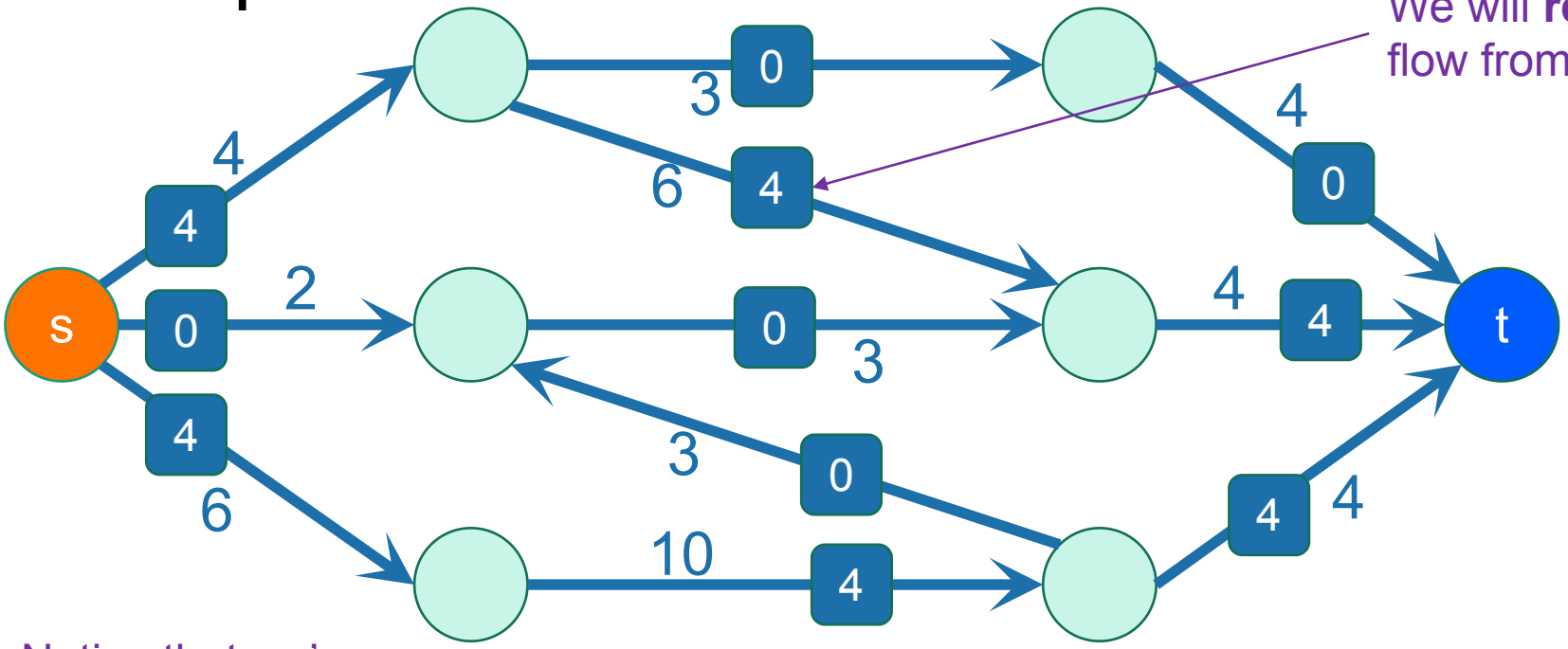


Example of Ford-Fulkerson

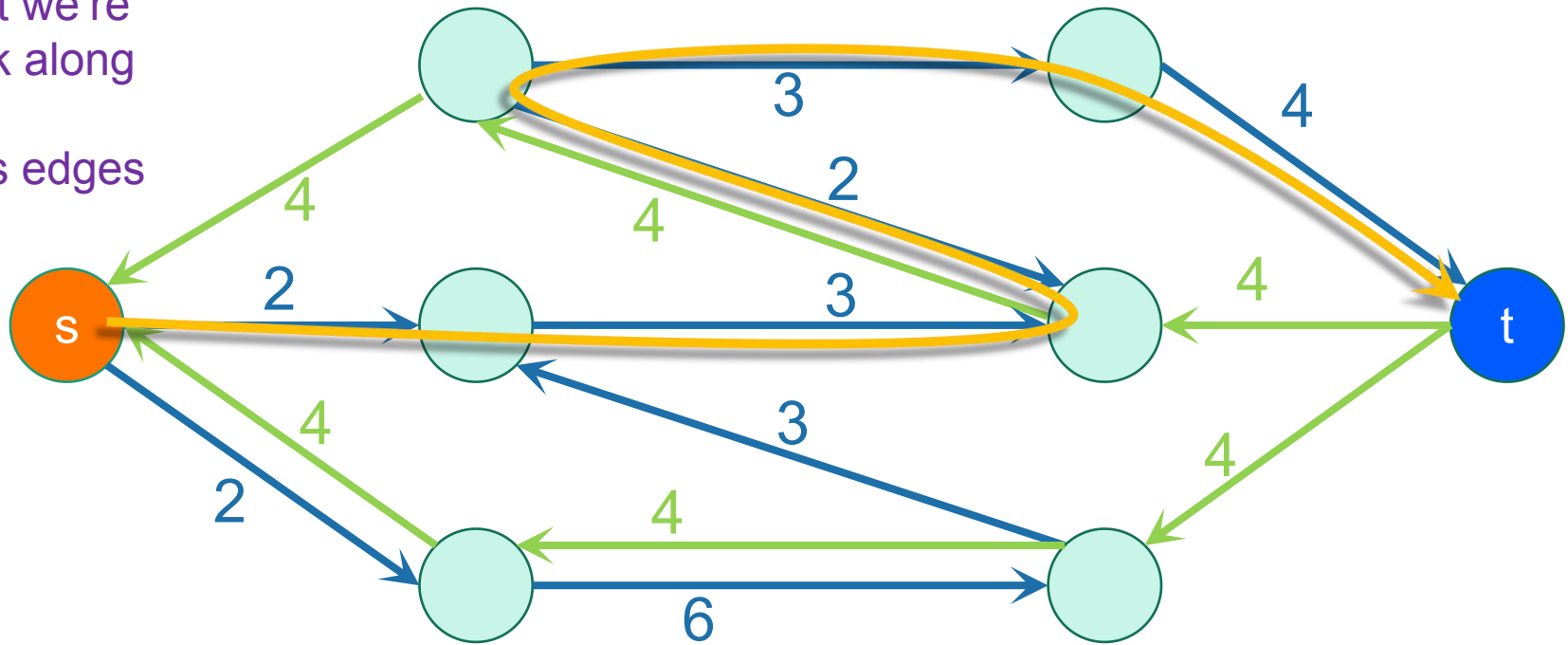




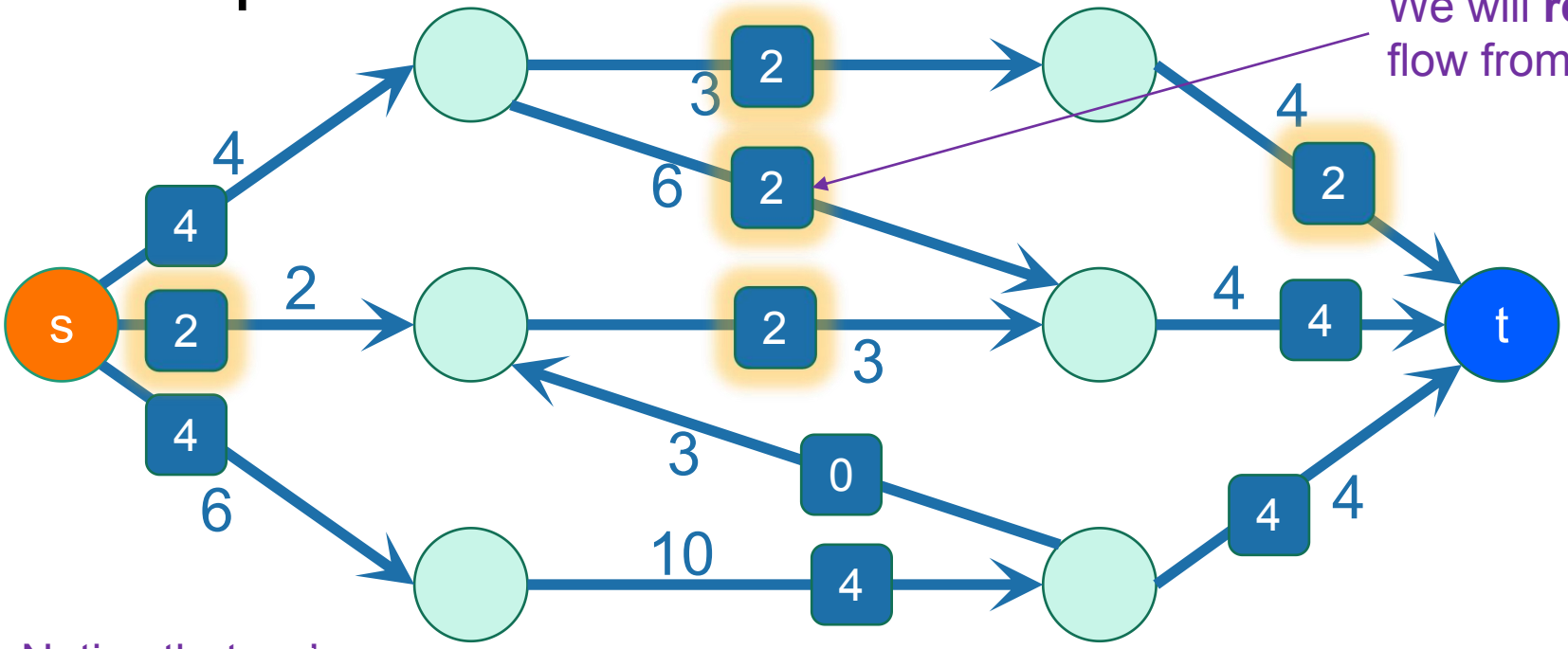
Example of Ford-Fulkerson



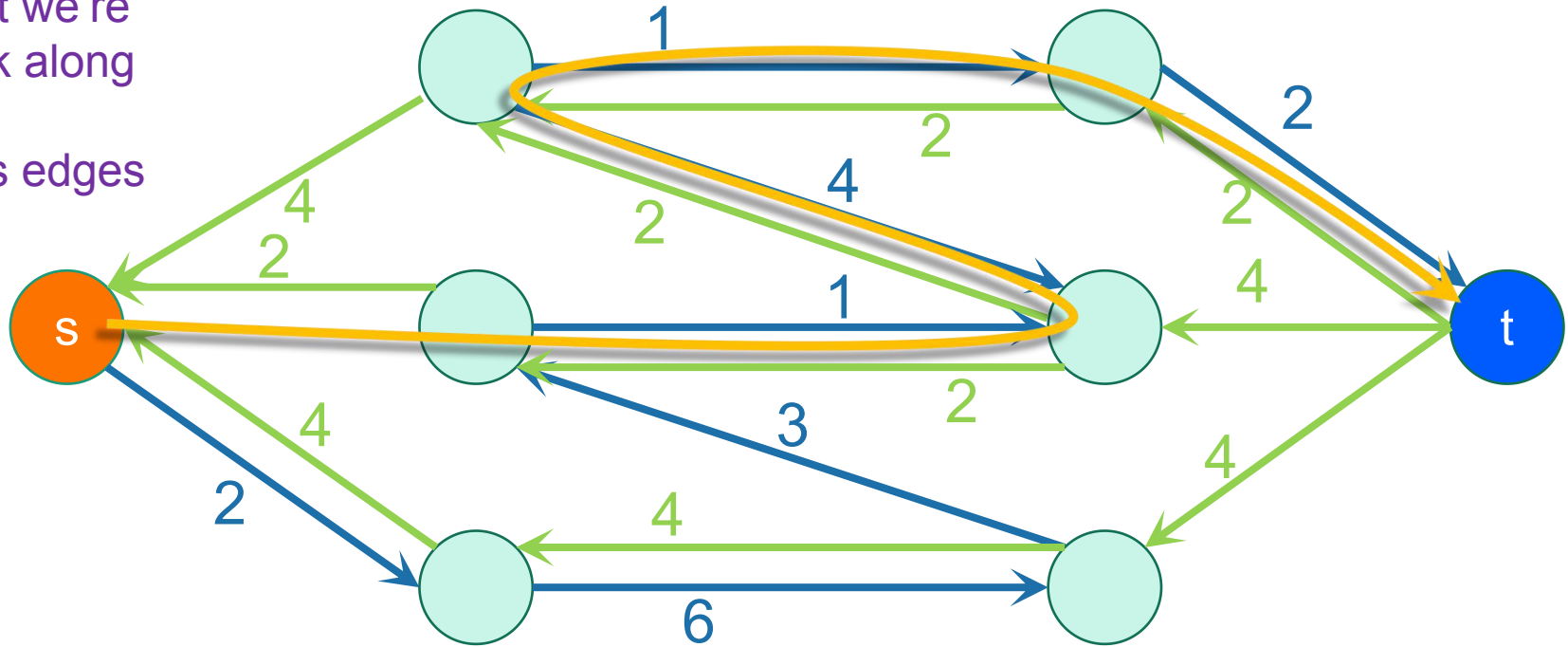
Notice that we're going back along one of the backwards edges we added.



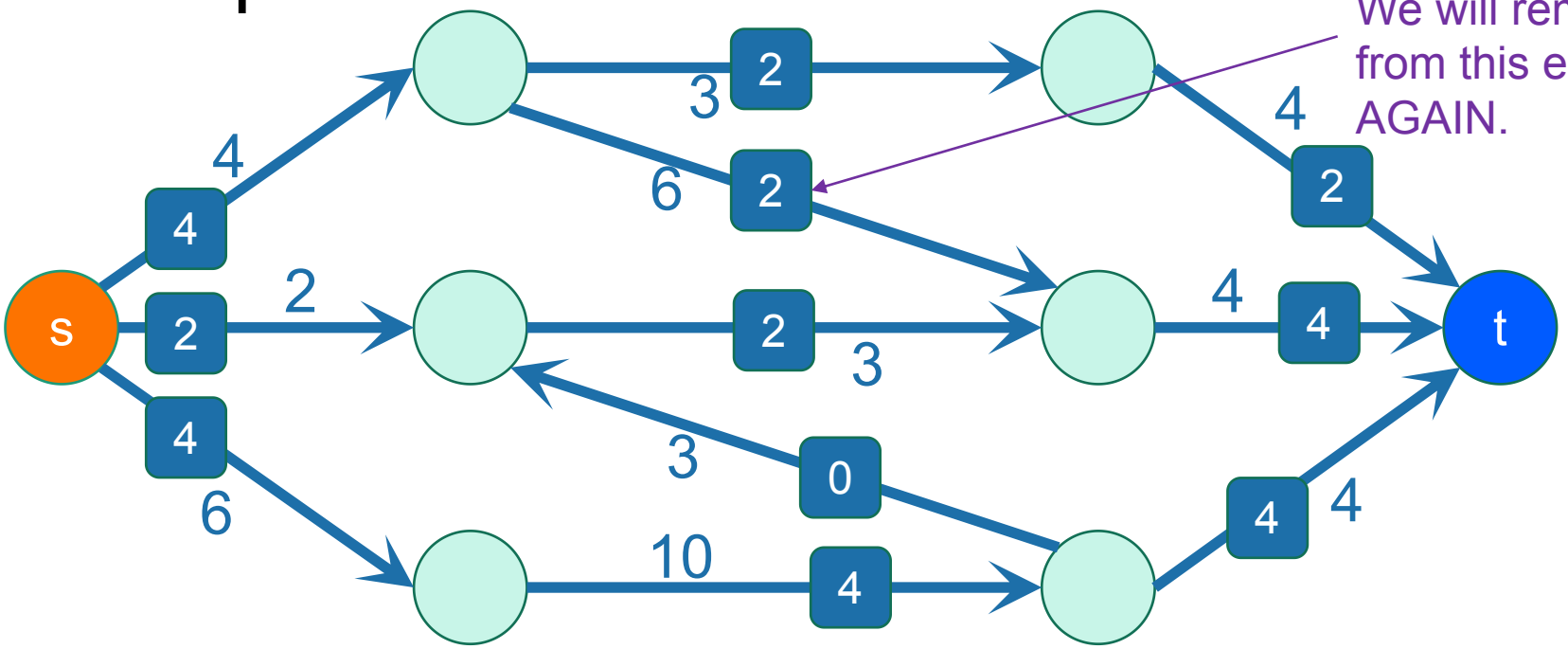
Example of Ford-Fulkerson

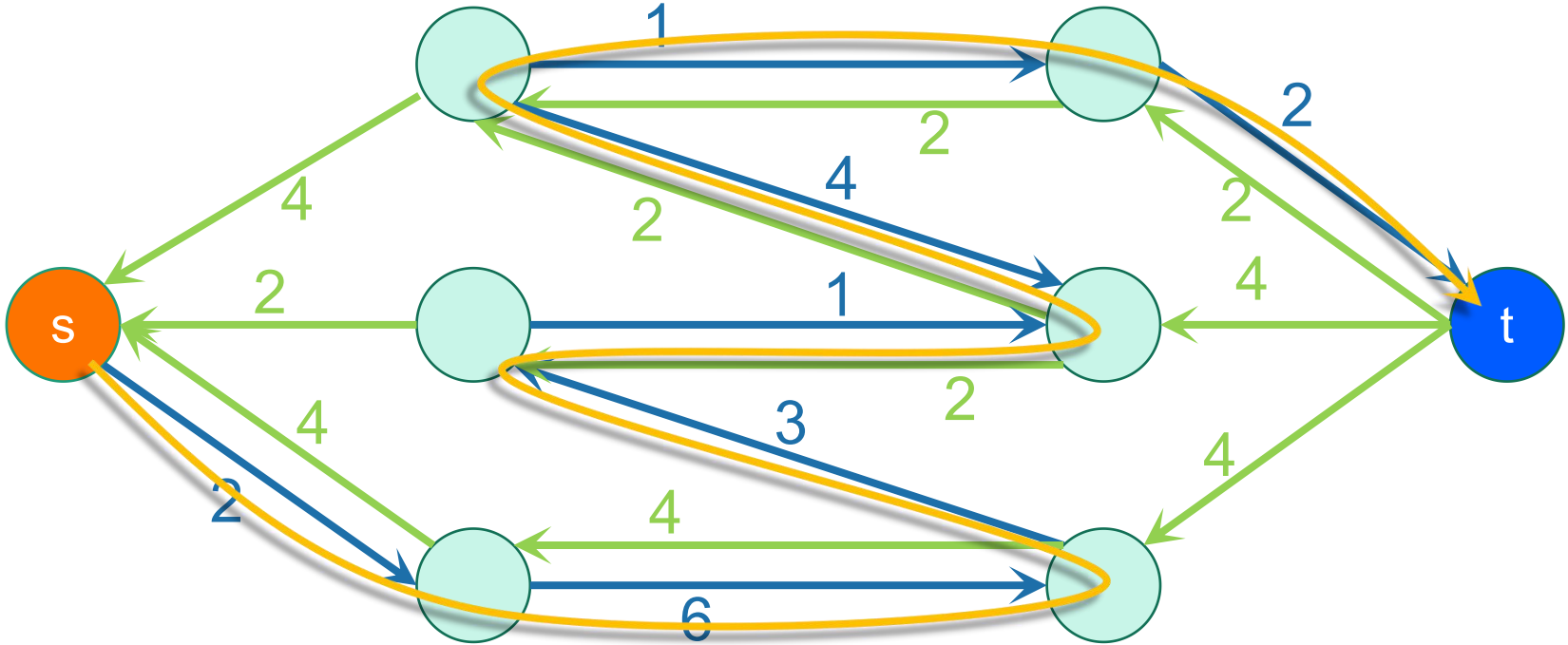


Notice that we're going back along one of the backwards edges we added.

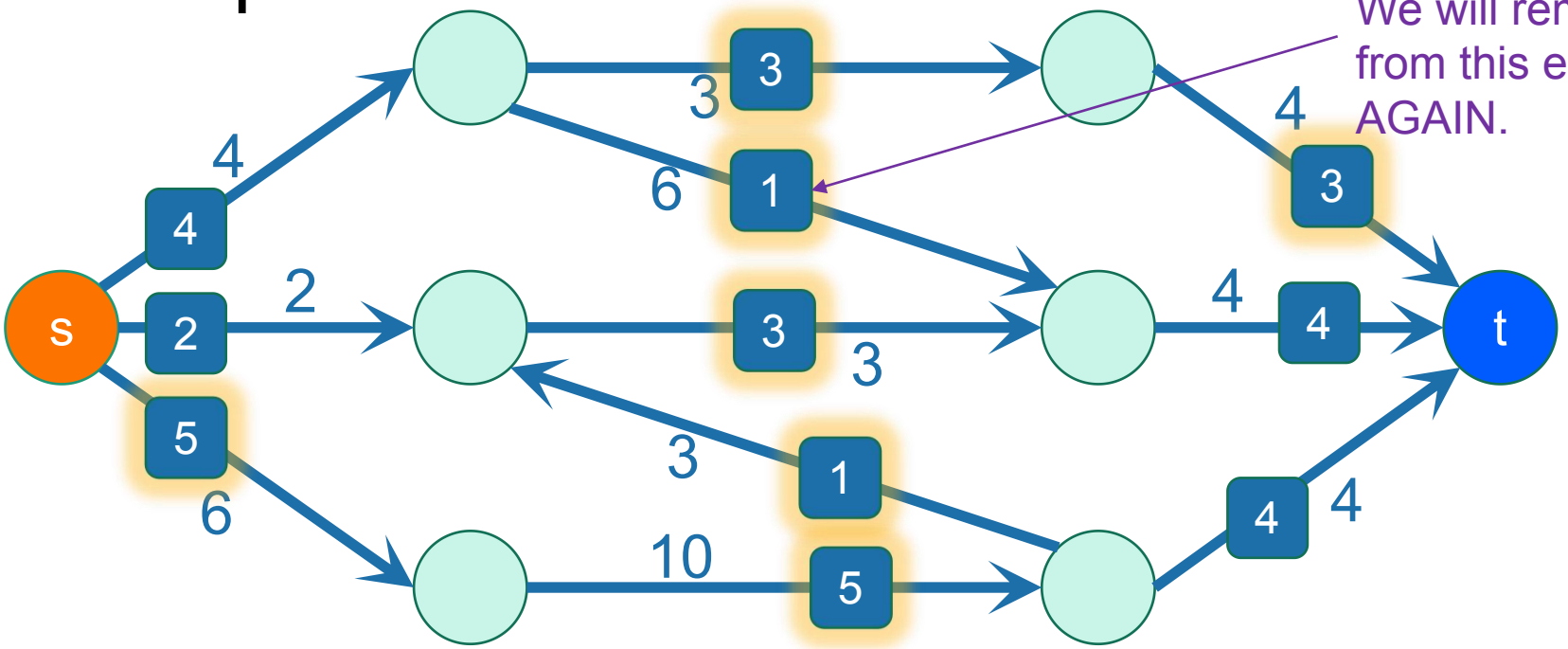


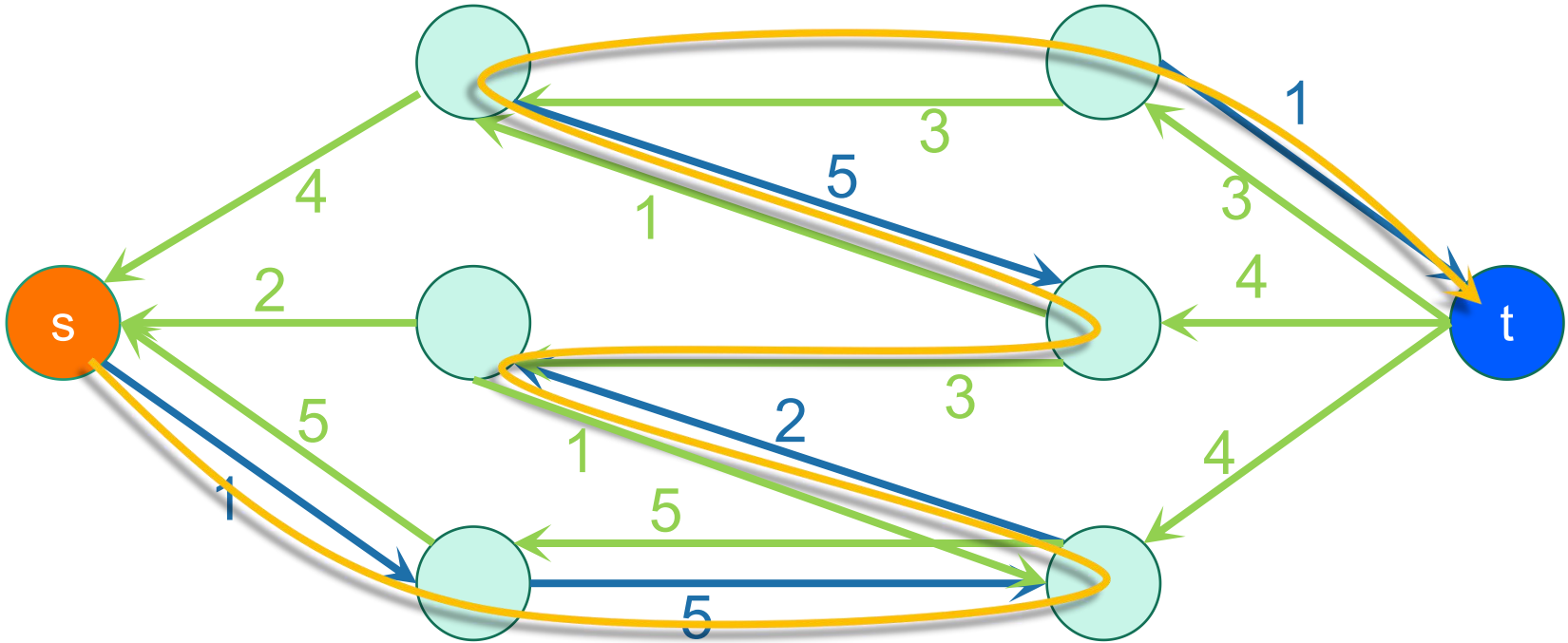
Example of Ford-Fulkerson



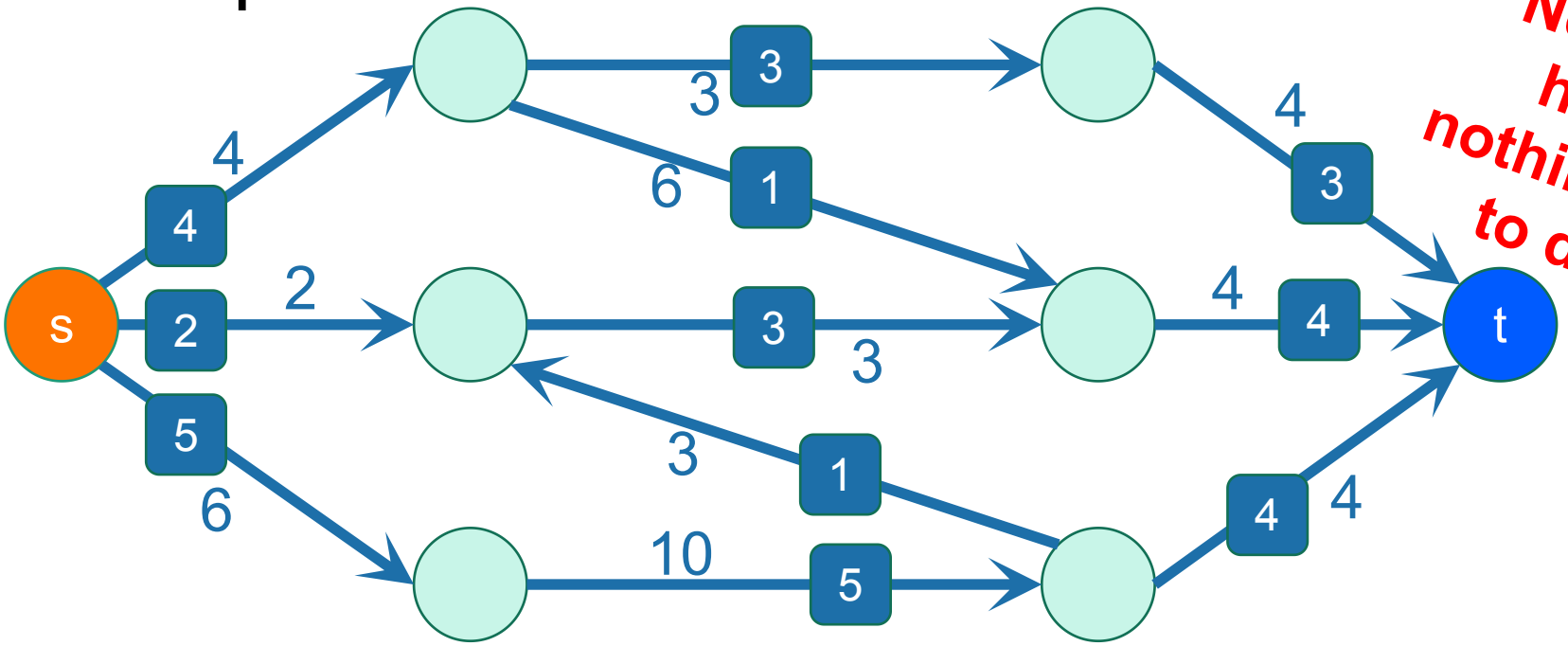


Example of Ford-Fulkerson

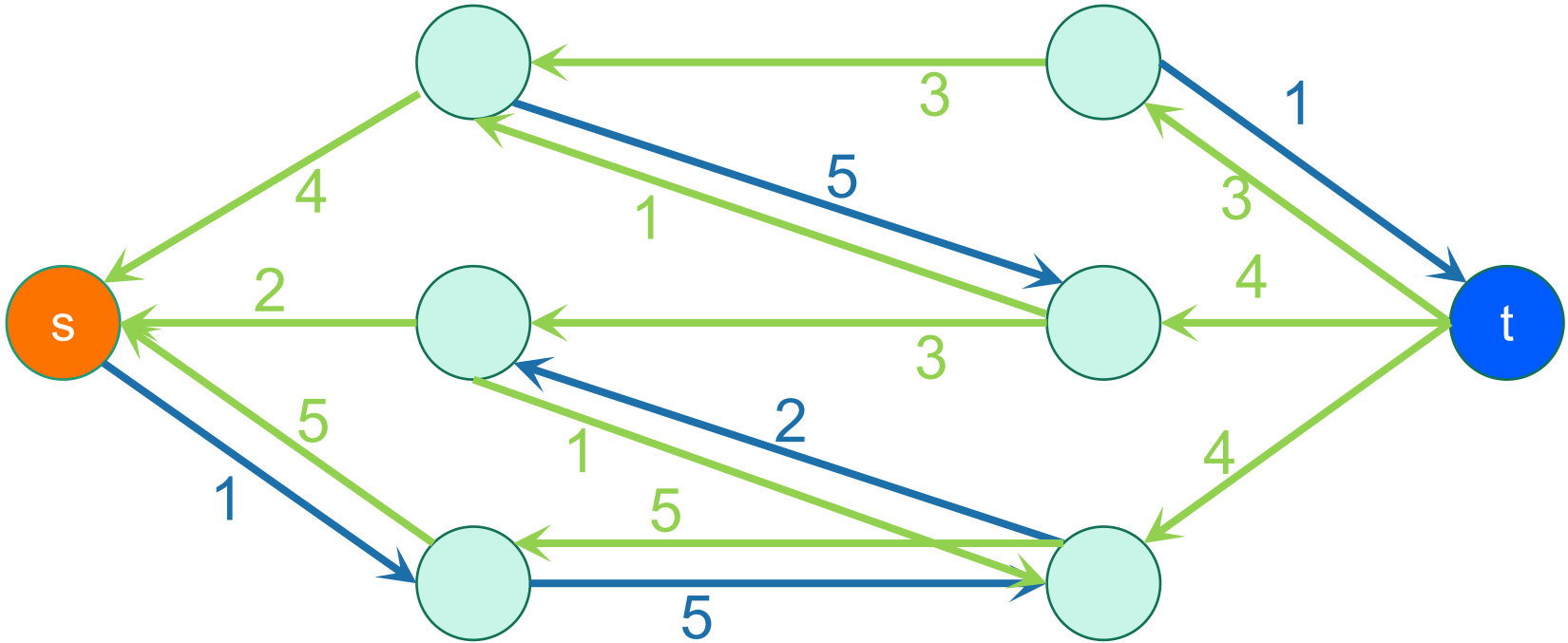




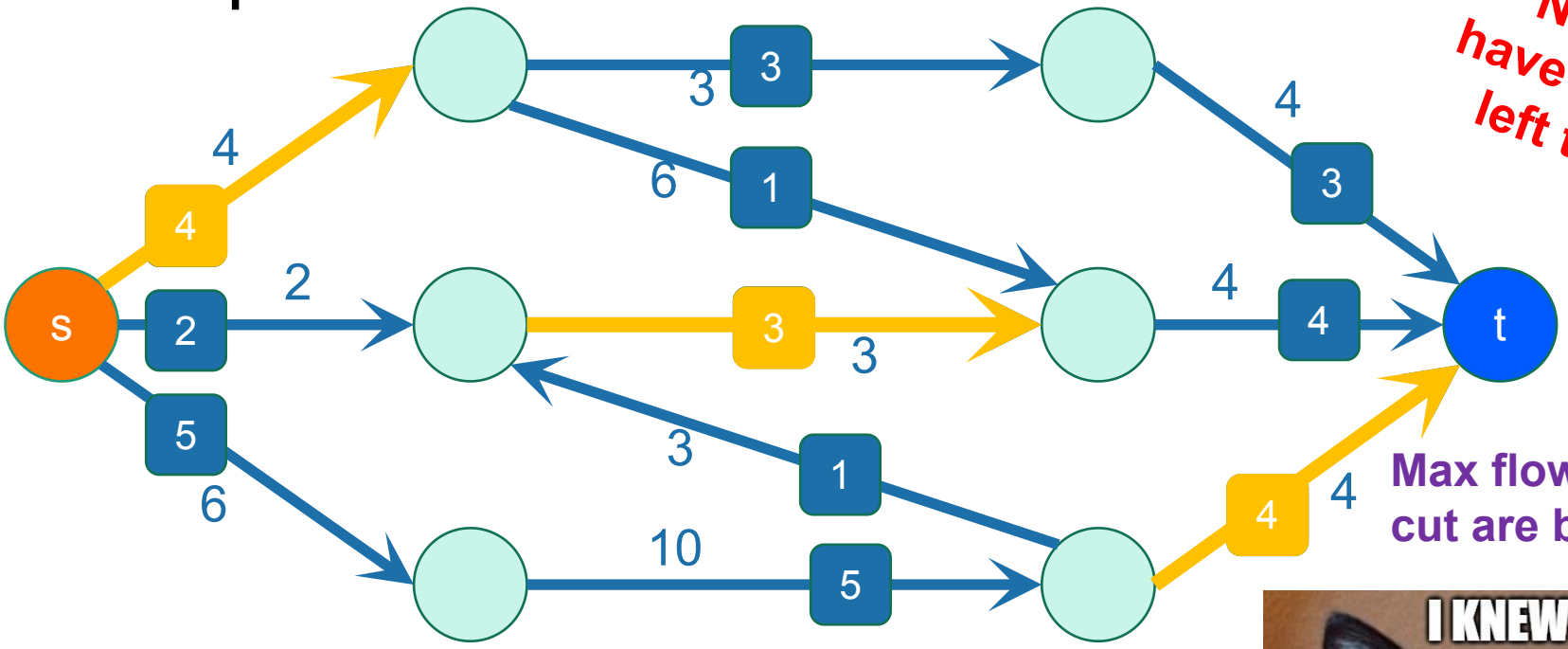
Example of Ford-Fulkerson



Now we have nothing left to do!

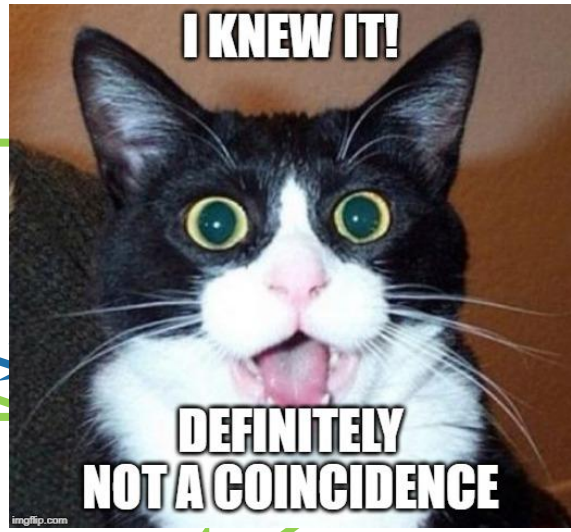


Example of Ford-Fulkerson



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???**