COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 18: Netflow Flow

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW6 Released!

Due 11/03 @ 11:59PM ET

HW5

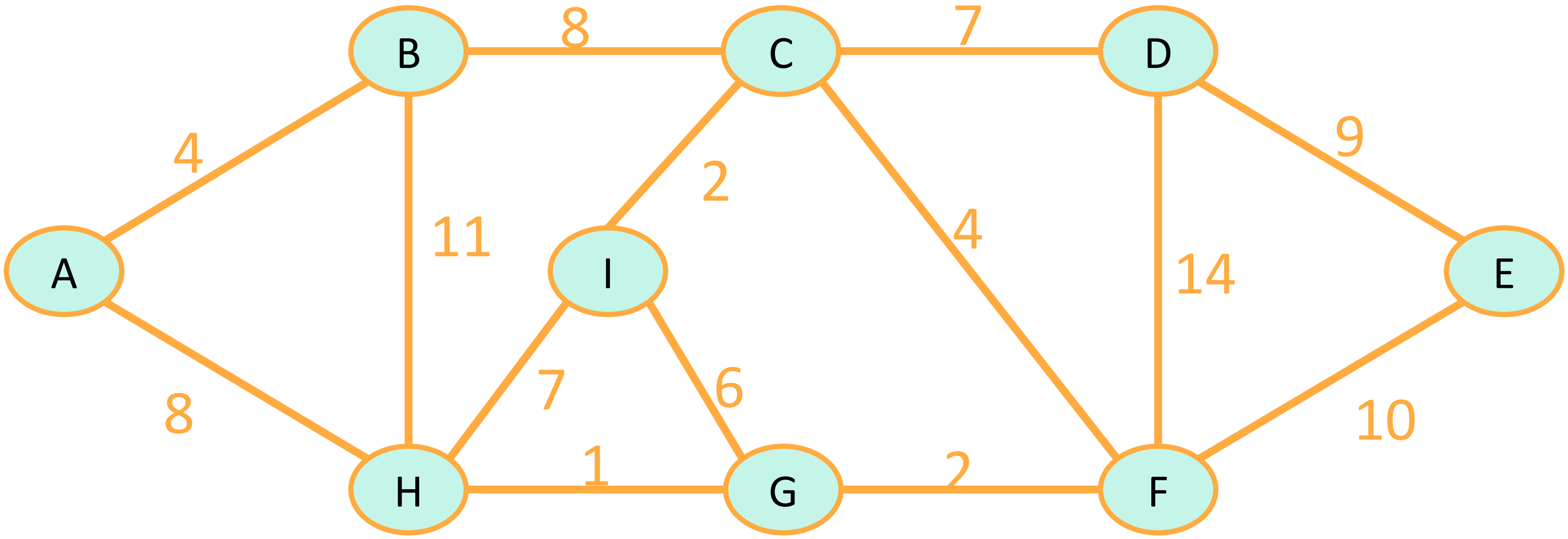
Grades released next week

Career Office Hours

- Denzel from Meta (sign ups here)
- Resume, general advice, behavioral interview practice, Meta, etc.

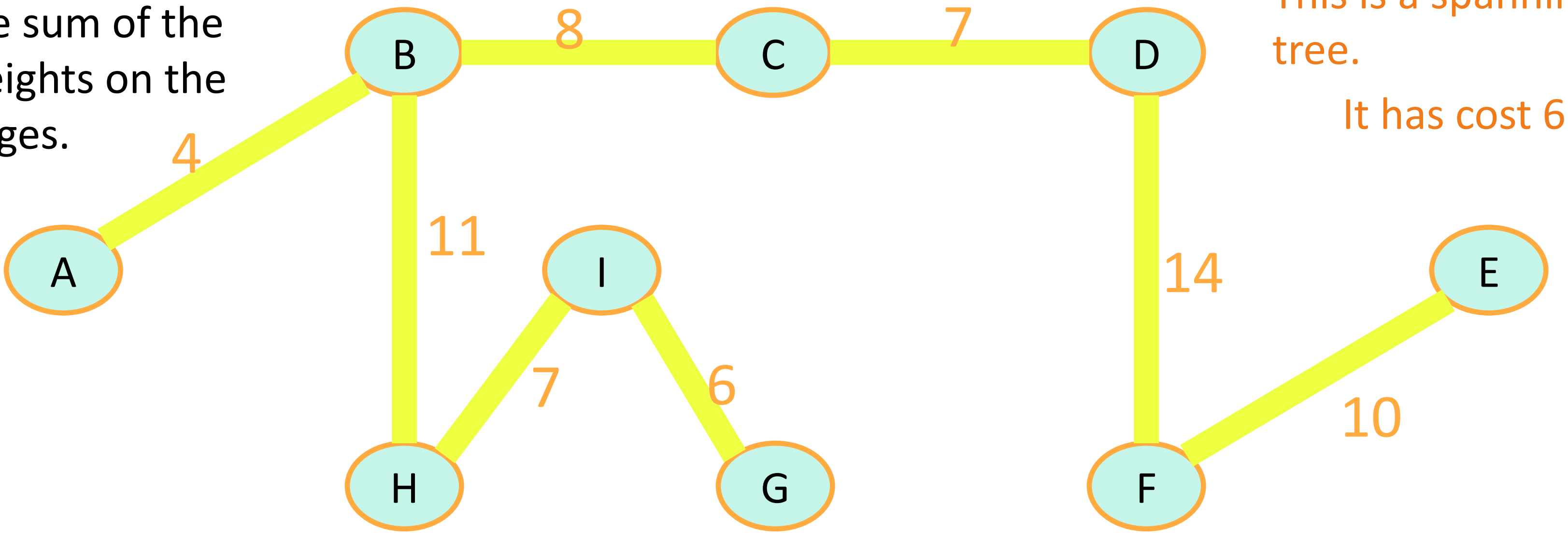
Recall where we ended last lecture…

What's a spanning tree?



What's a spanning tree?

The **cost** of a spanning tree is the sum of the weights on the edges.

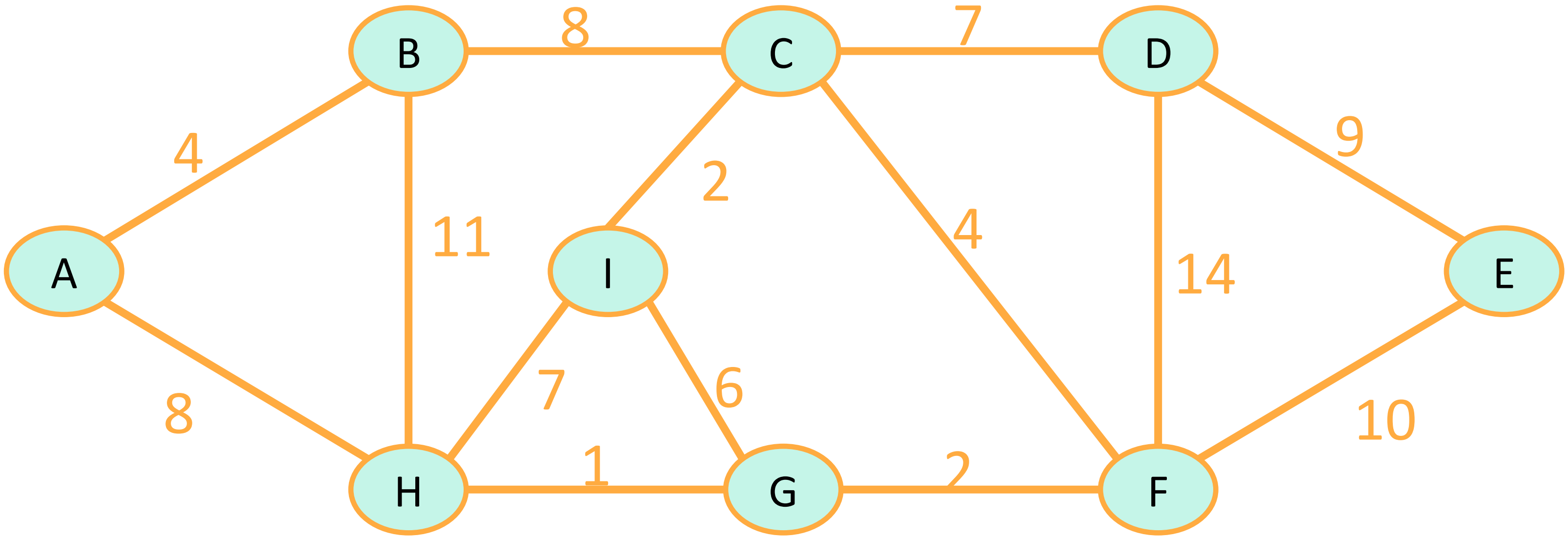


This is a spanning tree.

It has cost 67

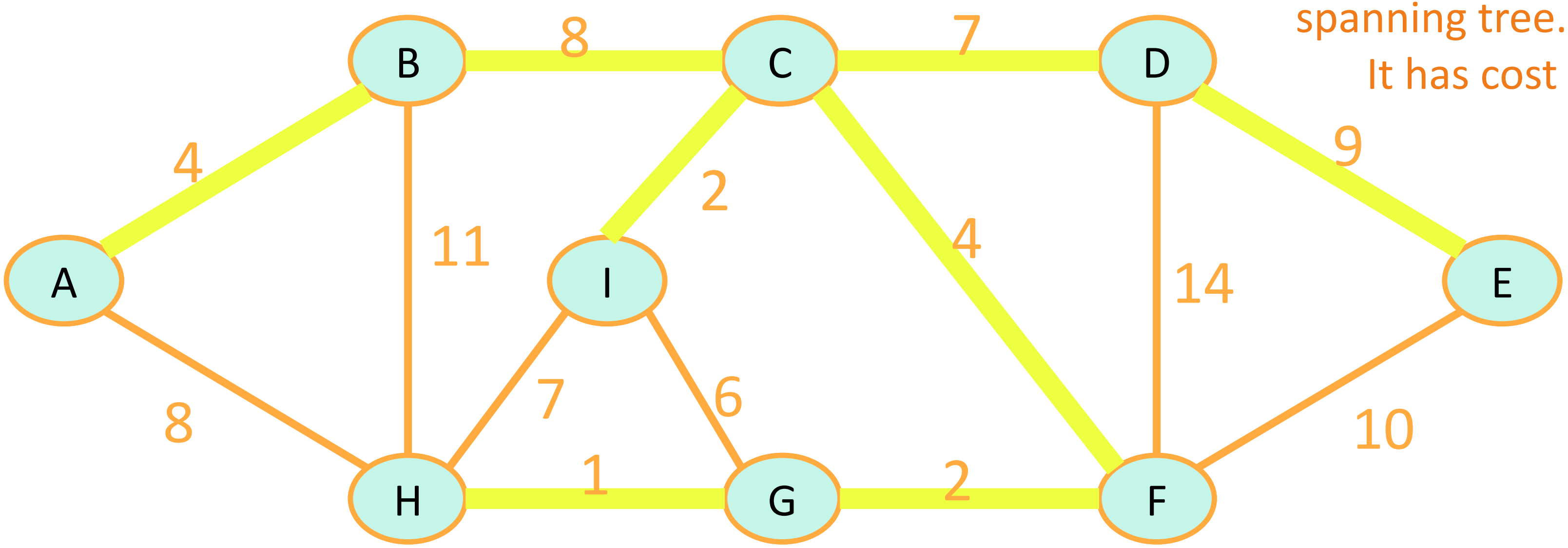
A **spanning tree** is a **tree** that connects all of the vertices (acyclic).

What's a Minimum Spanning Tree (MST)?



A **spanning tree** is a **tree** that connects all of the vertices (acyclic).

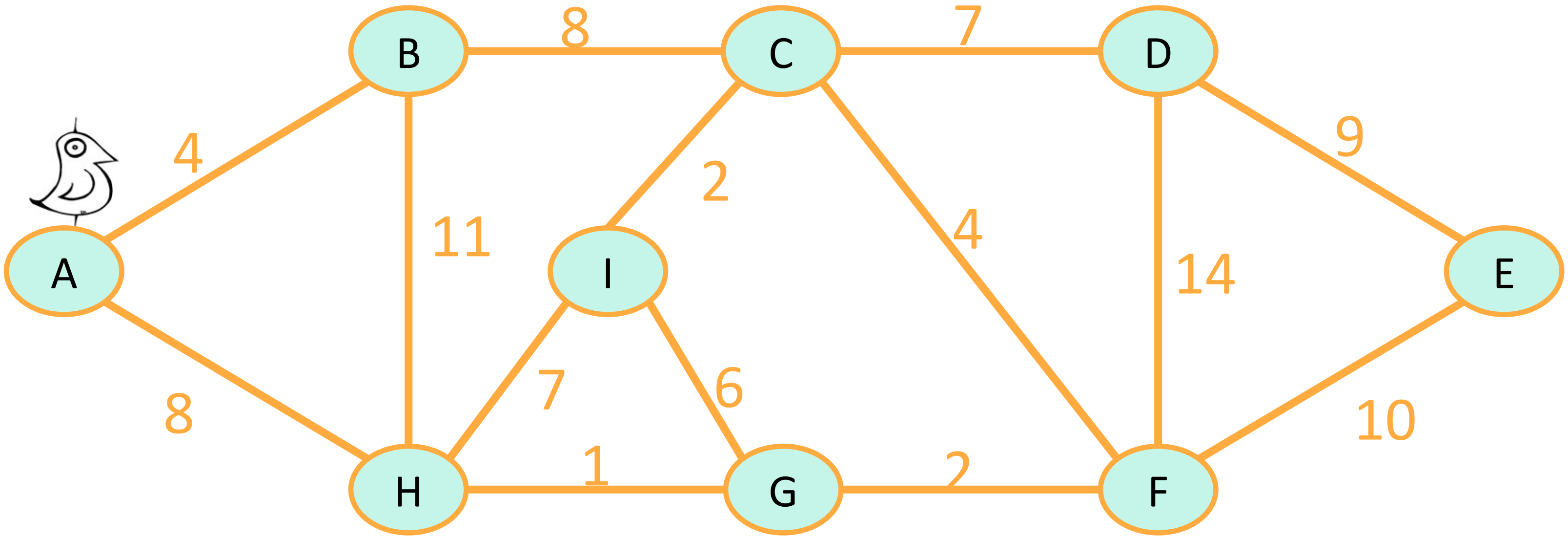
What's a Minimum Spanning Tree (MST)?



This is also a spanning tree.
It has cost 37

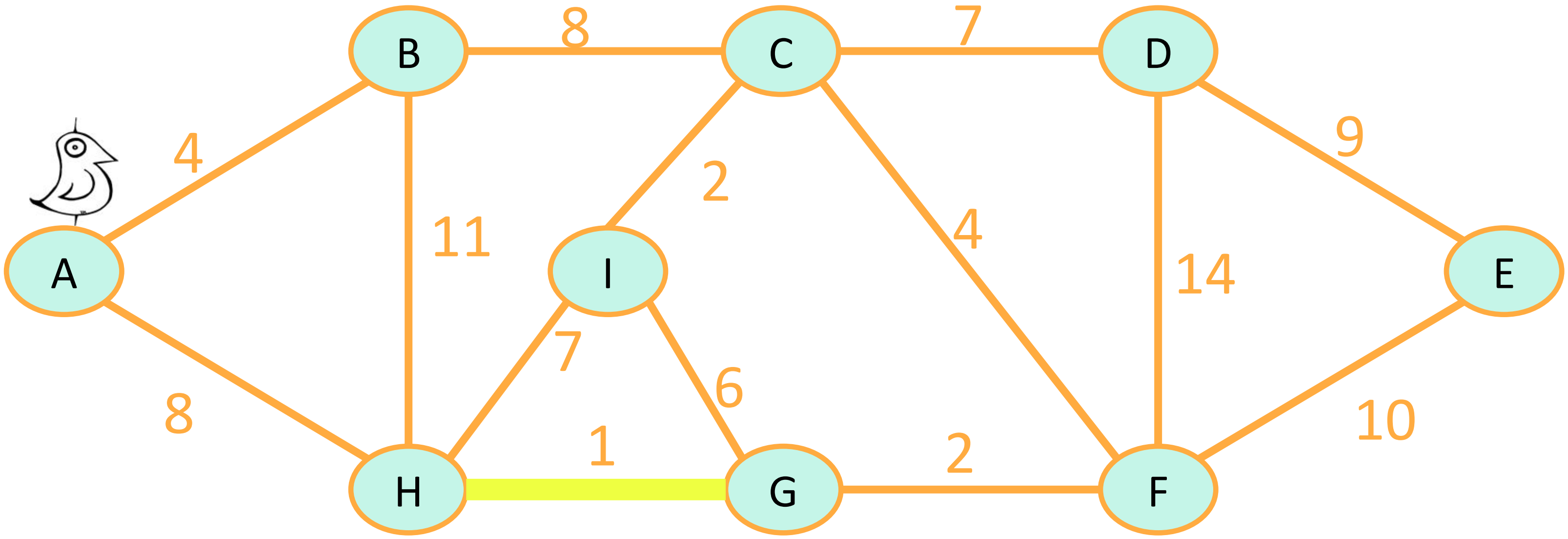
A **spanning tree** is a **tree** that connects all of the vertices (acyclic).

Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.

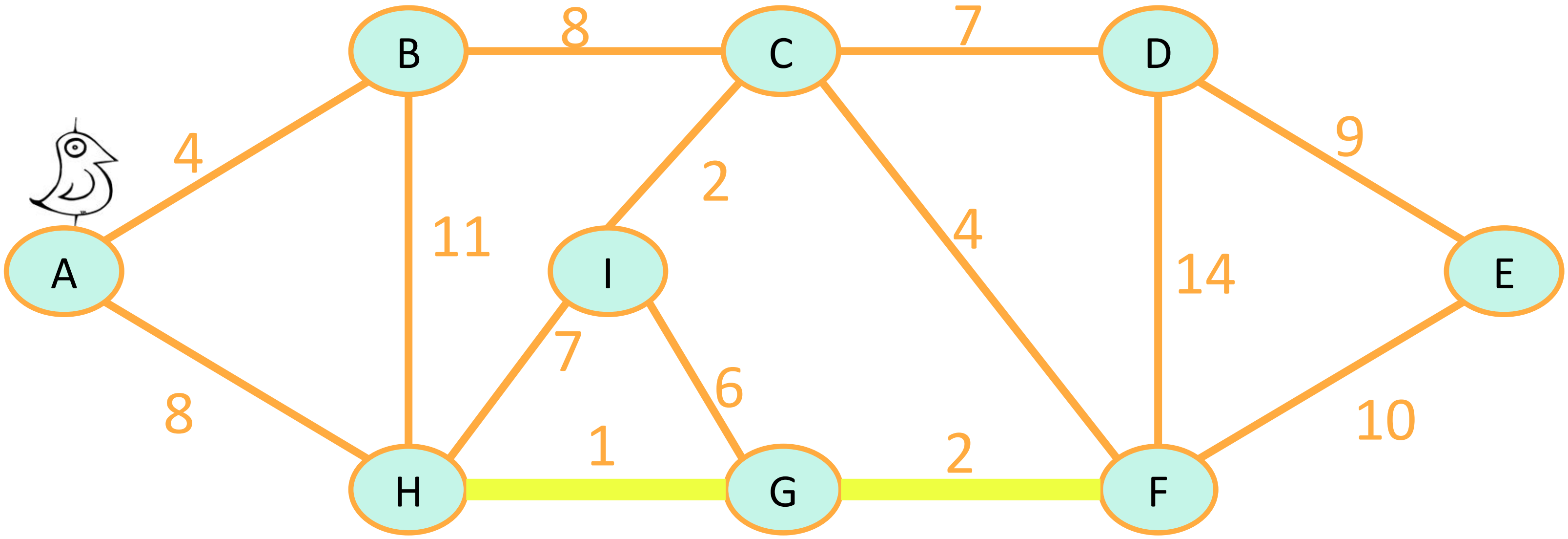


Start at node H!

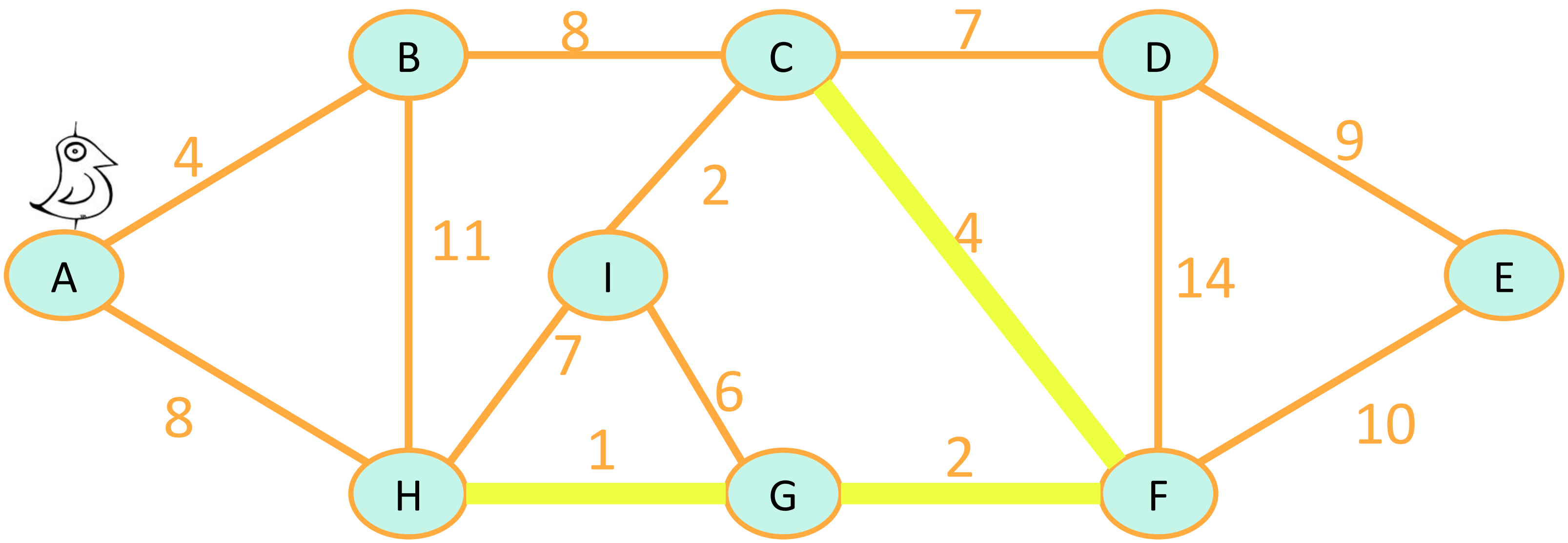
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



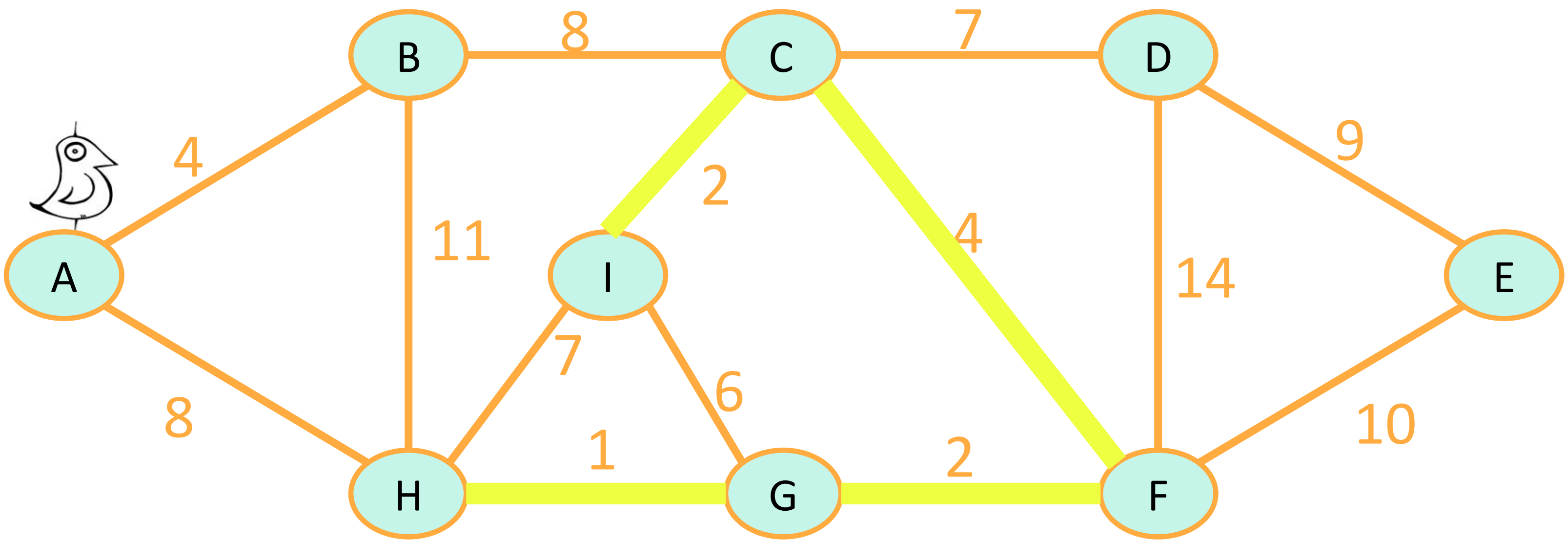
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



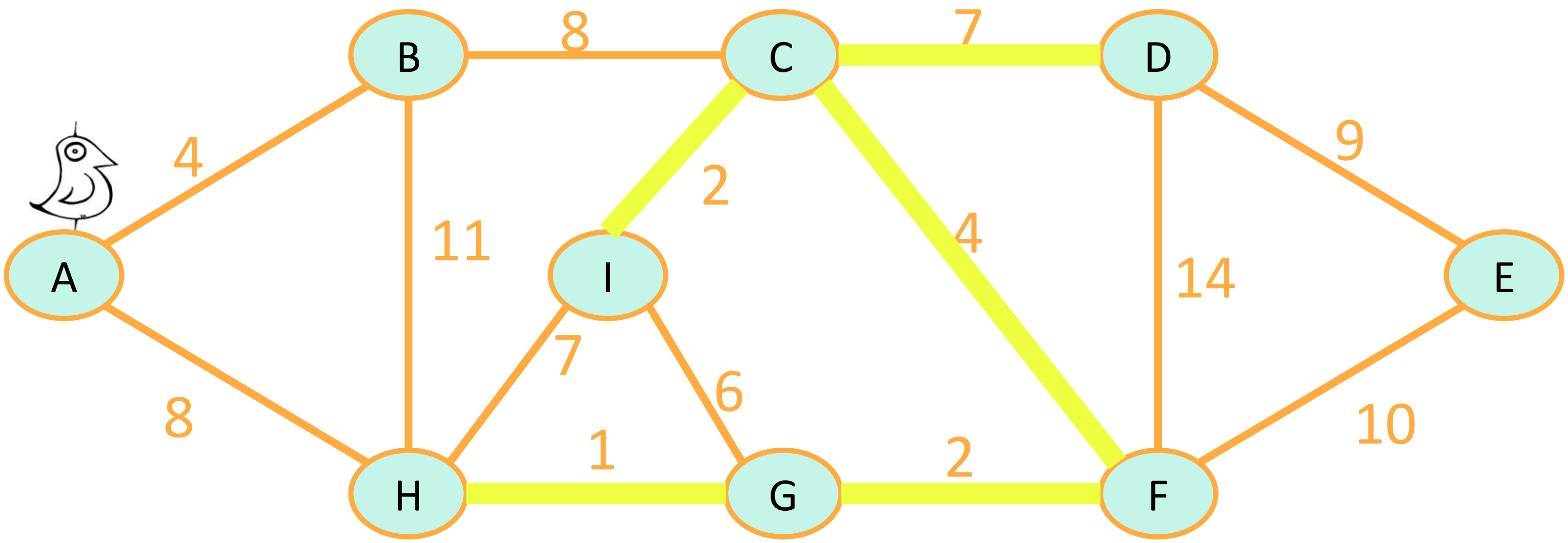
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



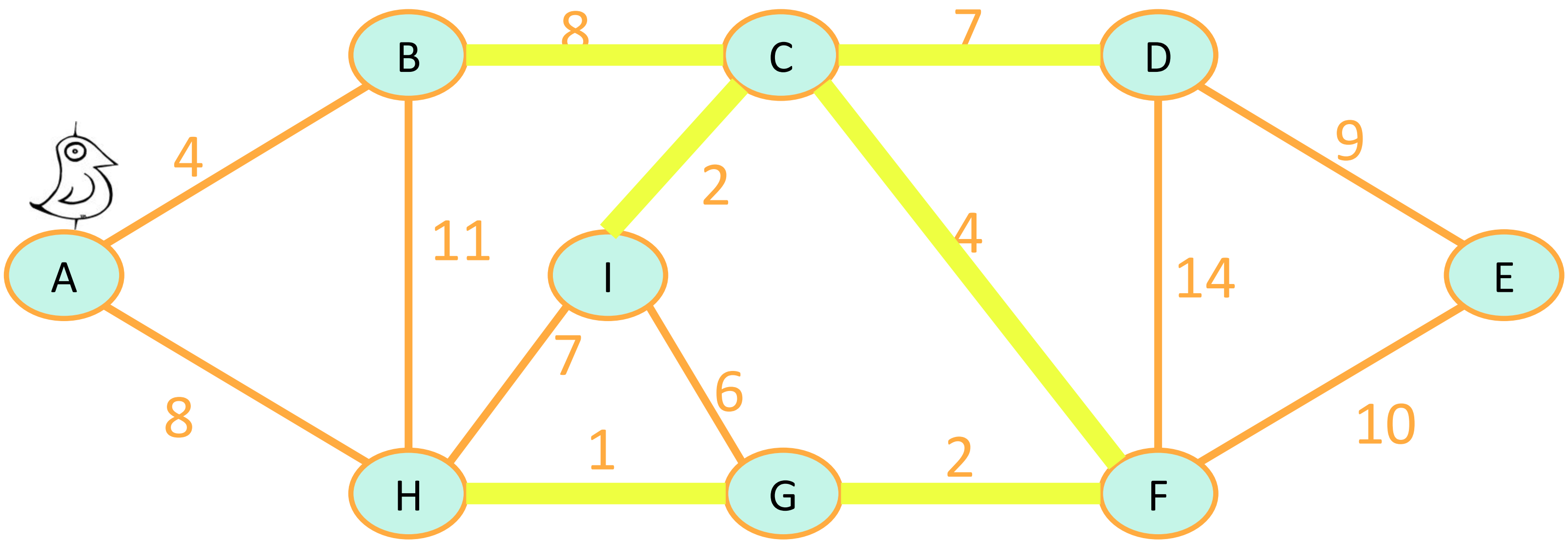
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



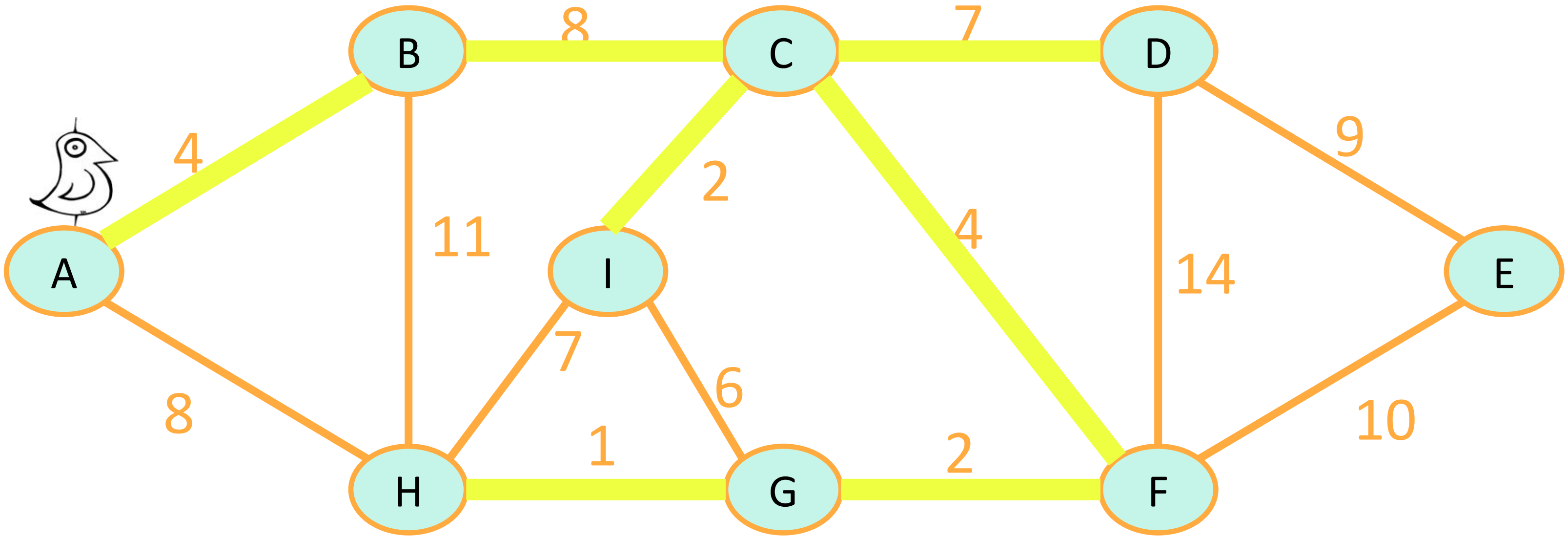
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



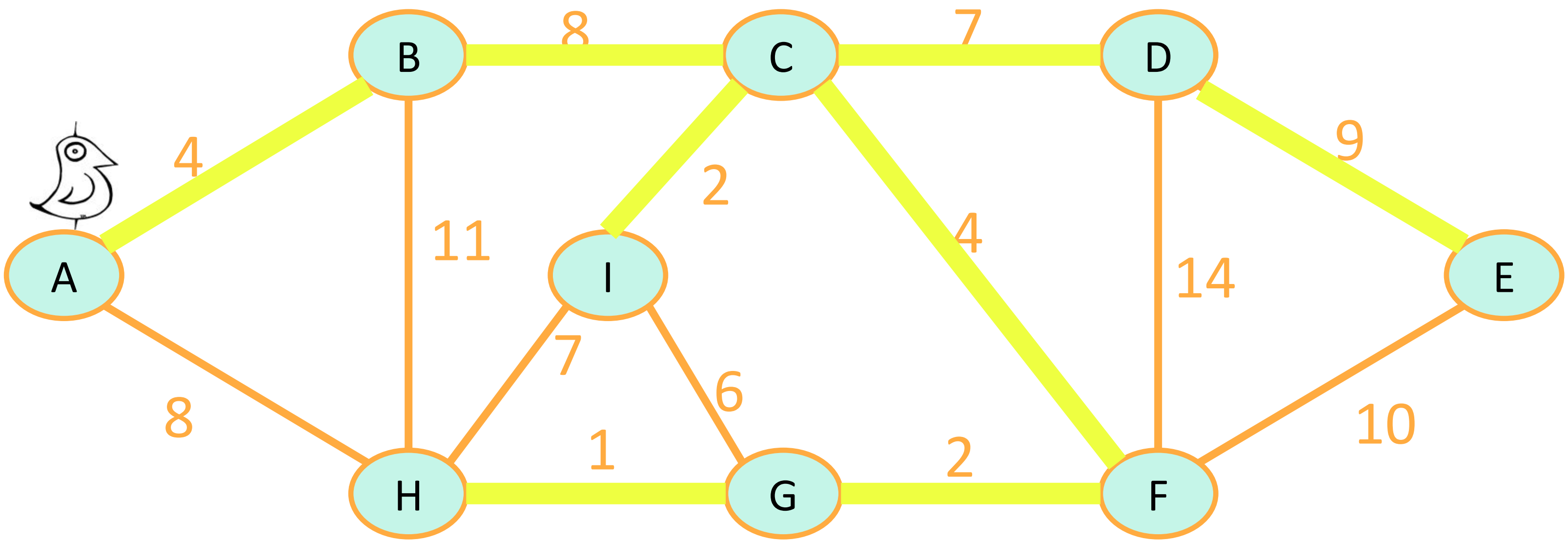
Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



Idea 1: Start growing a tree, (greedily) add the shortest edge we can to grow the tree.



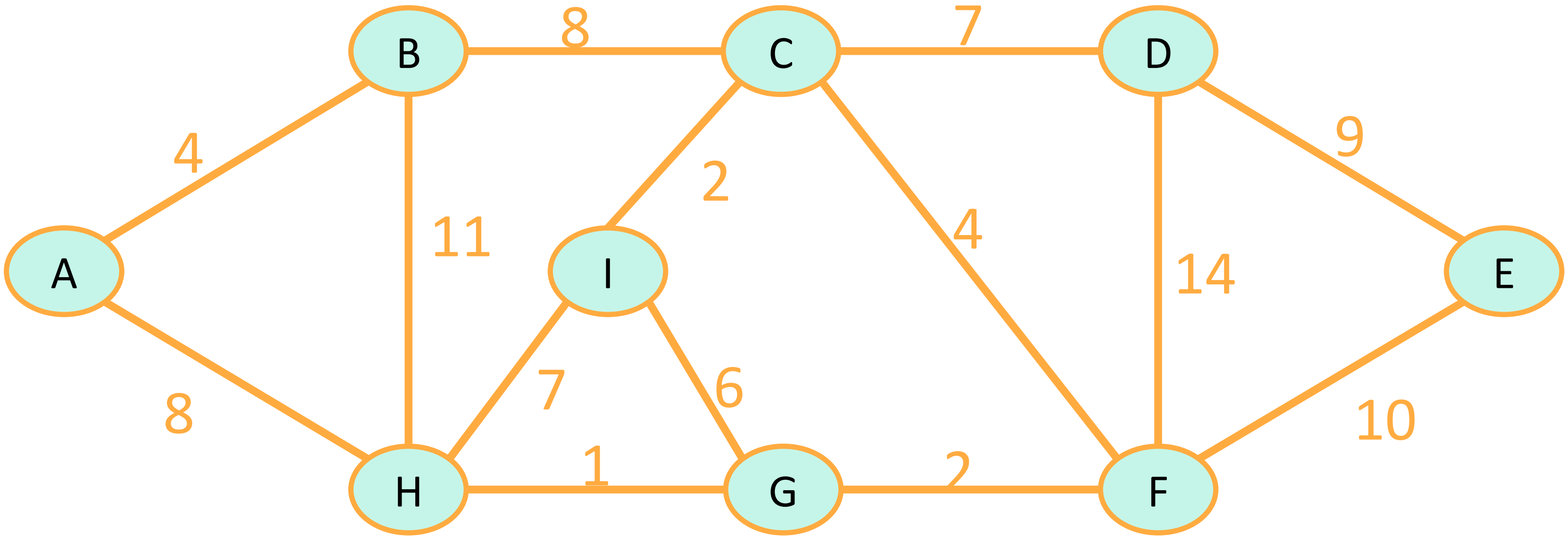
We've Discovered Prim's Algorithm!

- Prim(G = (V,E), starting vertex s):
  - Let (s,u) be the lightest edge coming out of s.
  - MST = { (s,u) }
  - verticesVisited = { s, u }
  - **while** |verticesVisited| < |V|:
    - find the lightest edge {x,v} in E so that:
      - x is in verticesVisited
      - v is not in verticesVisited
    - add {x,v} to MST
    - add v to verticesVisited
  - **return** MST

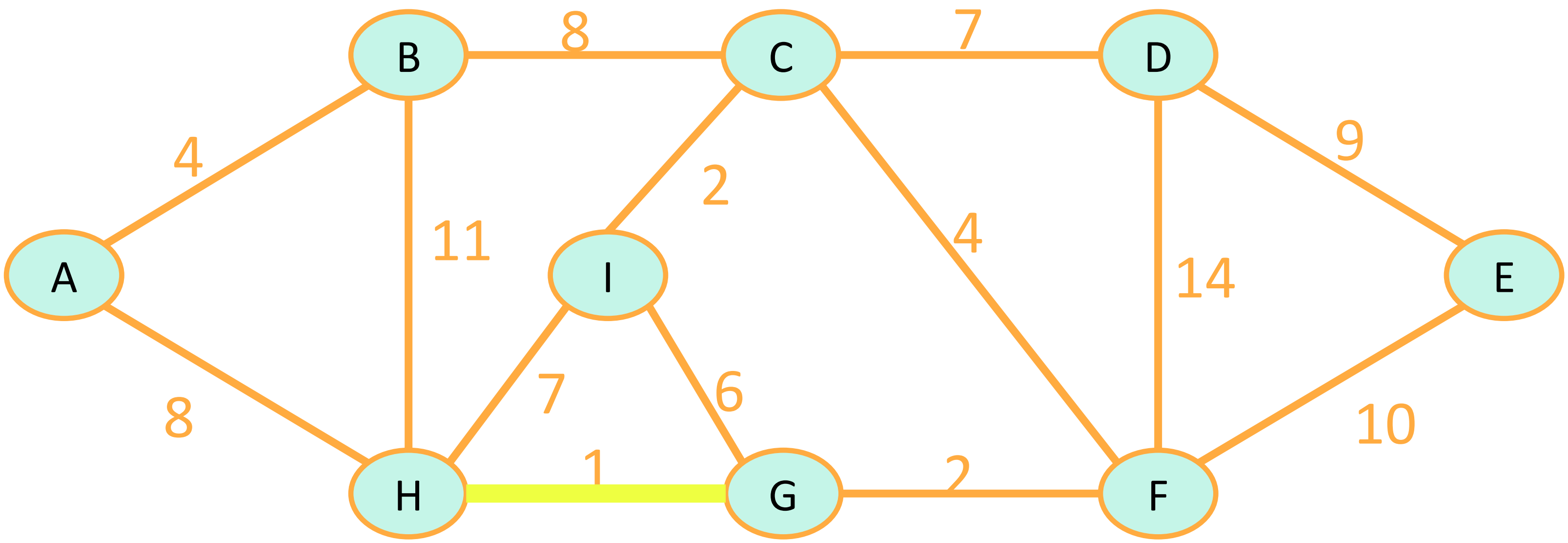
At most V iterations of this while loop.

Time at most E to go through all the edges and find the lightest.

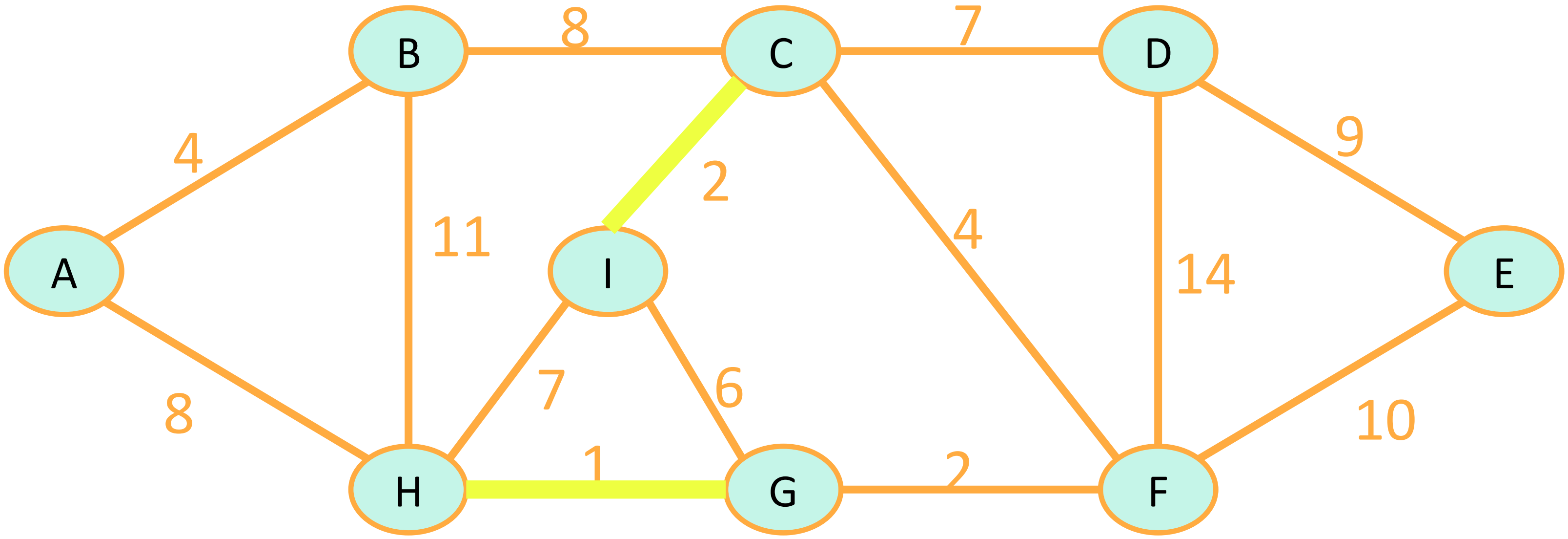
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



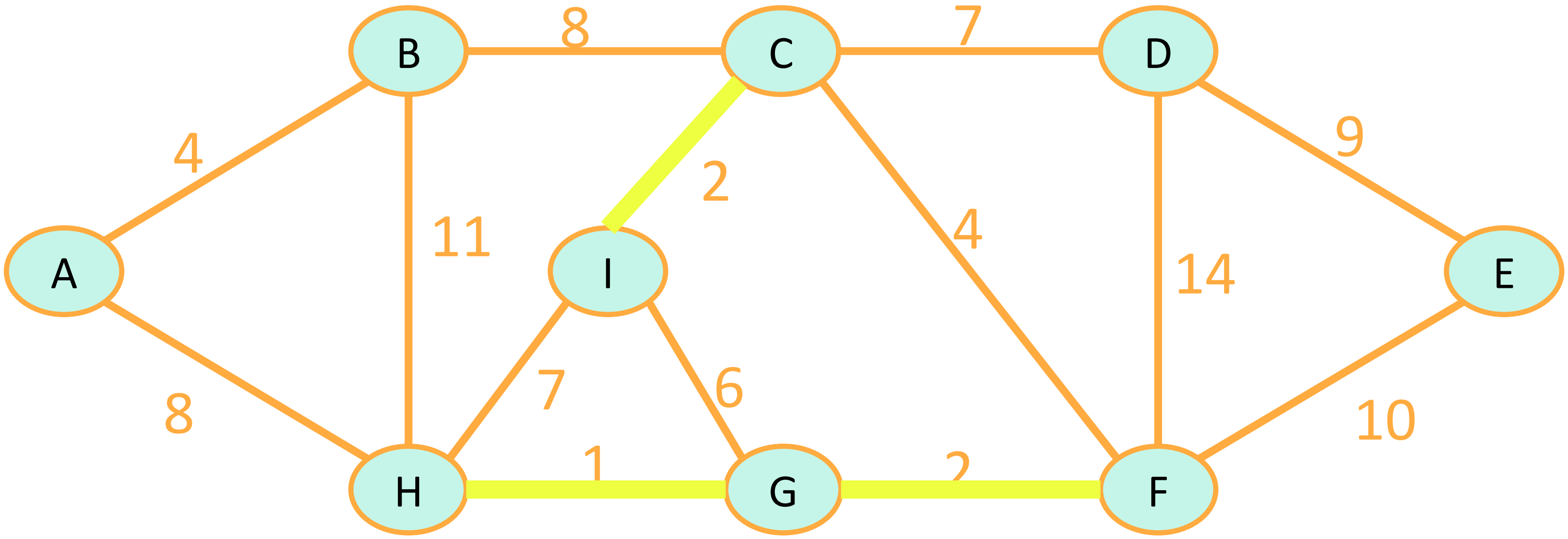
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



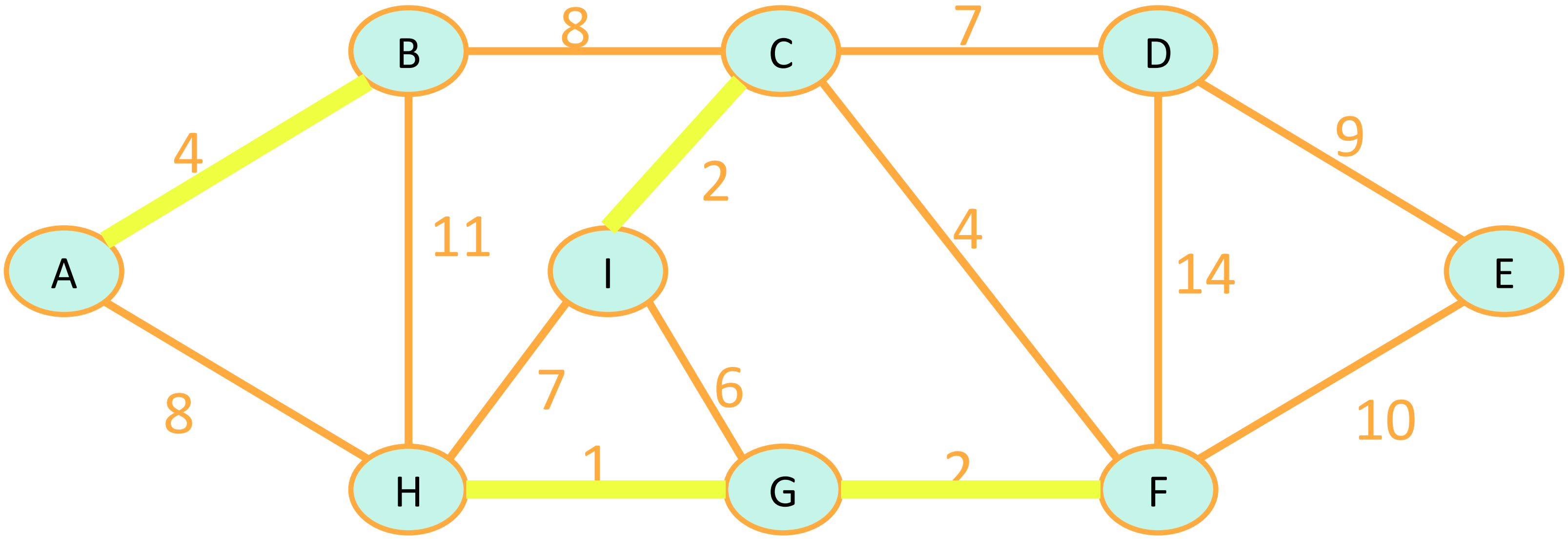
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?



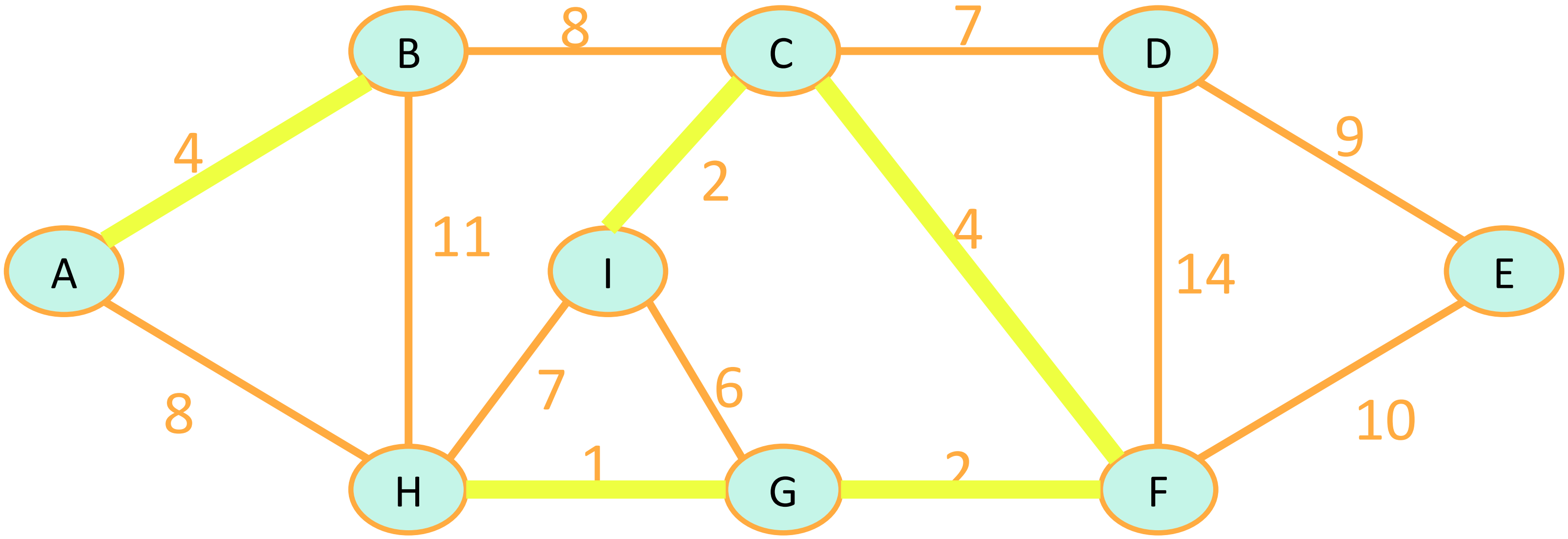
Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?



Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

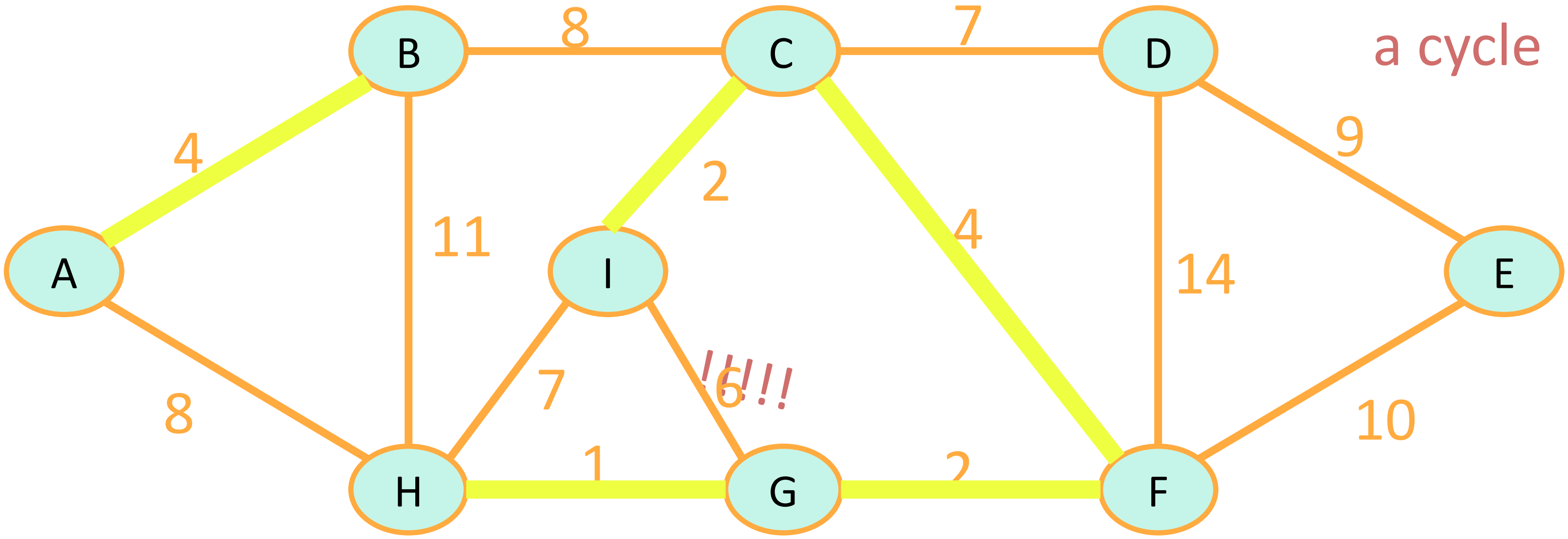


Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?



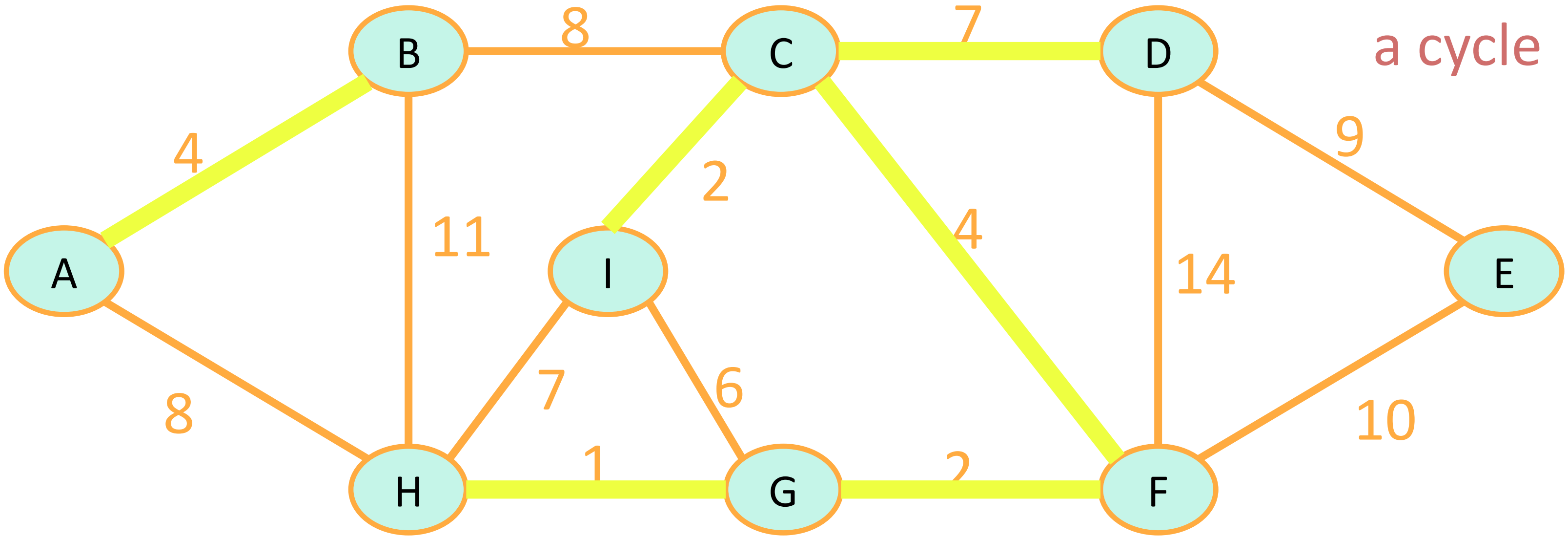
Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

That won't cau
a cycle



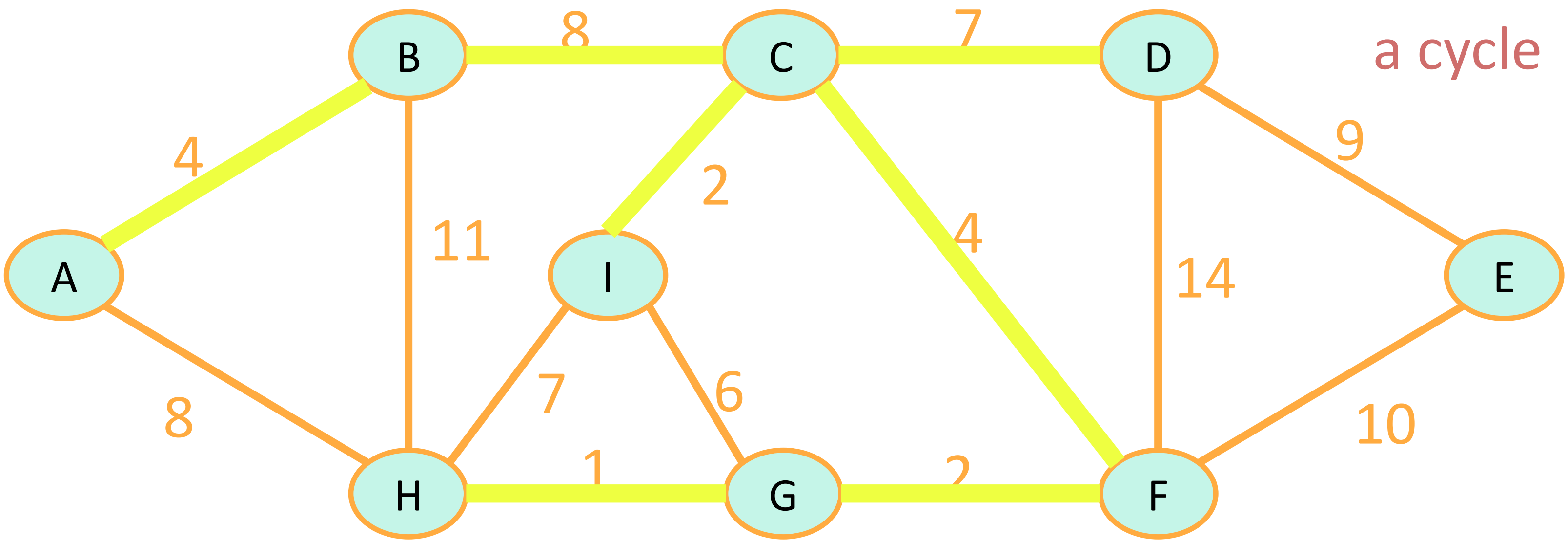
Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

That won't ca
a cycle



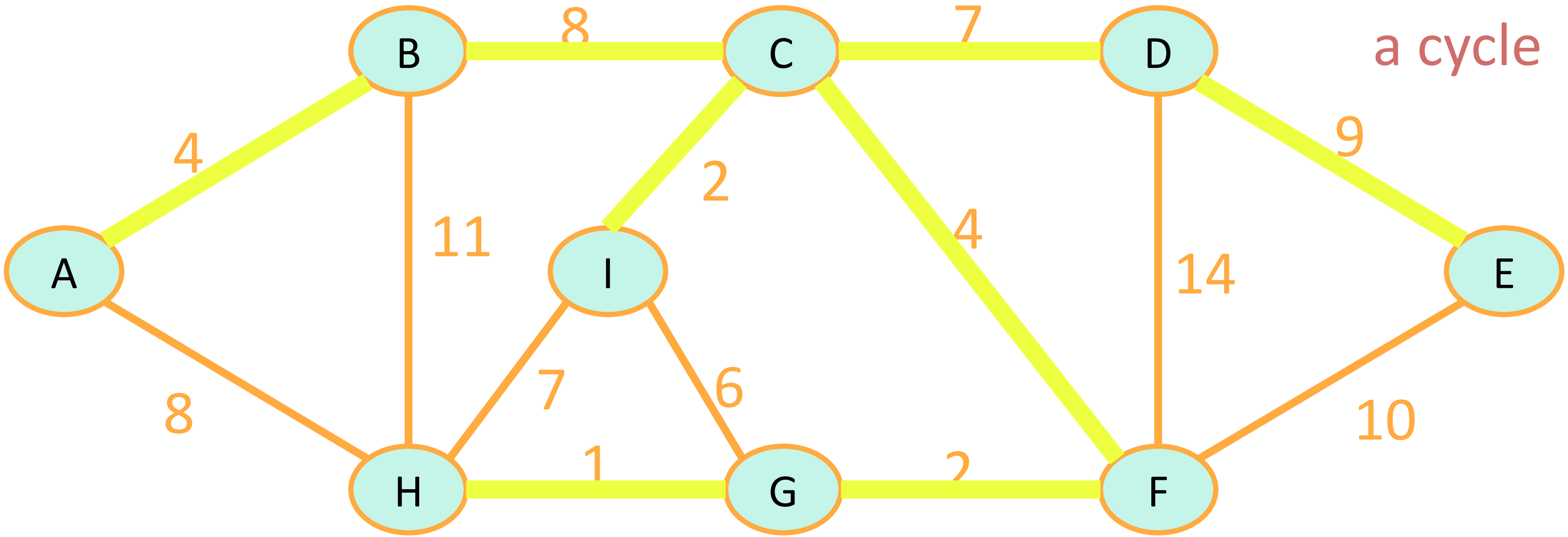
Idea #2: What if we just always take the cheapest edge? Whether or not it's connected to what we have so far?

That won't ca
a cycle



Idea #2: What if we just always take the cheapest edge?
Whether or not it's connected to what we have so far?

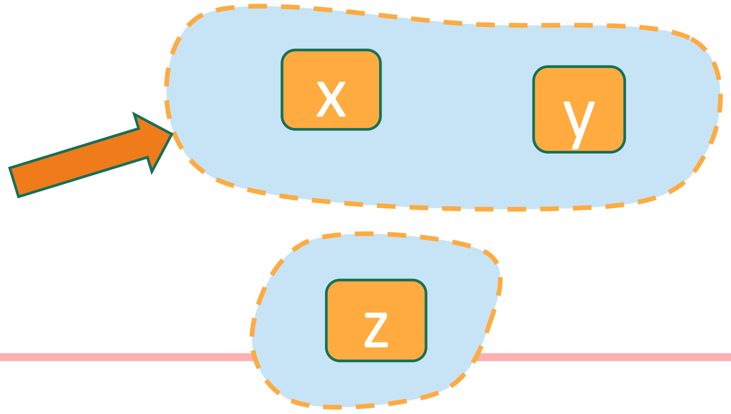
That won't cau
a cycle



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

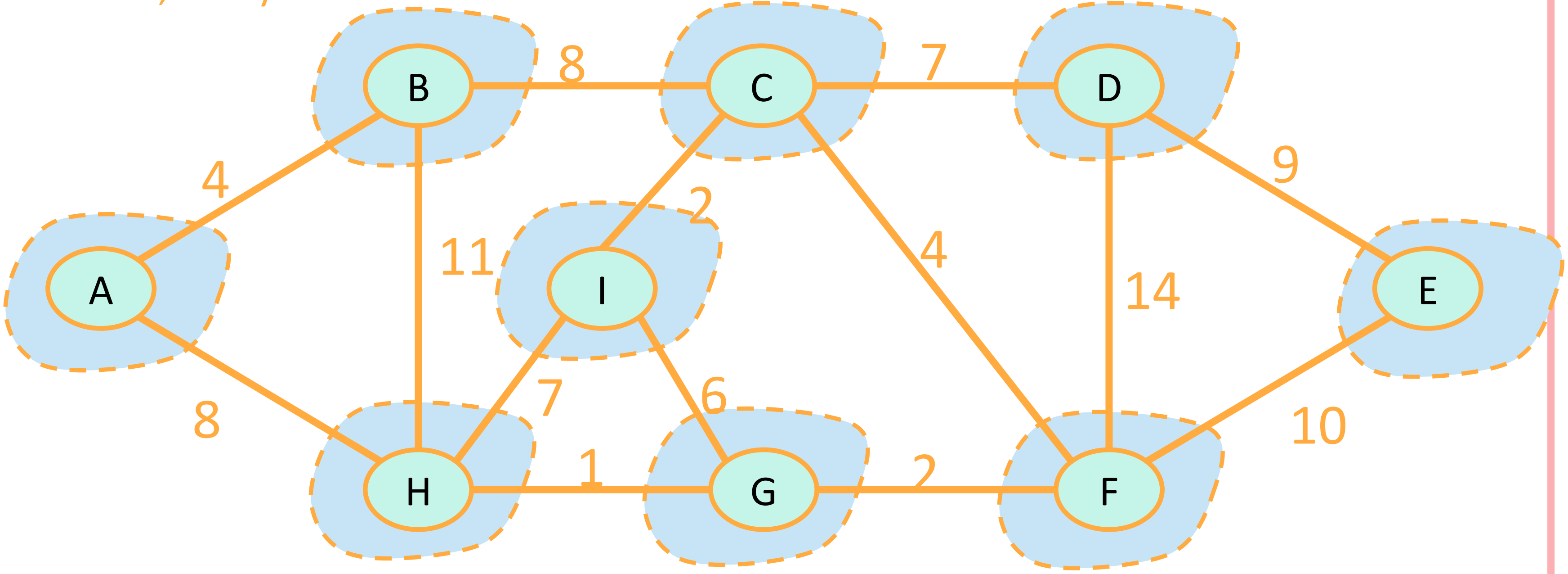
- Used for storing collections of sets
- Supports:
  - **makeSet(u):** create a set {u}
  - **find(u):** return the set that u is in
  - **union(u,v):** merge the set that u is in with the set that v is in.

```
makeSet(x)
makeSet(y)
makeSet(z)
union(x,y)
find(x)
```



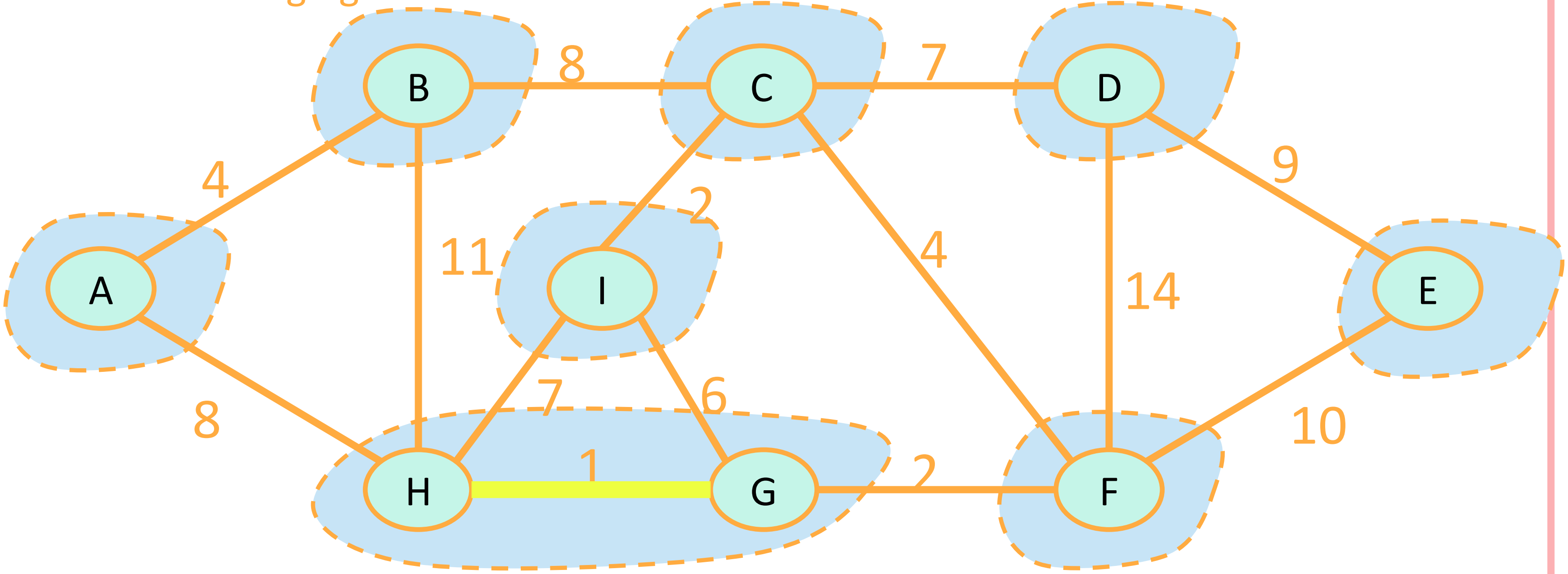
At each step of Kruskal's, we maintain a forest (collection of trees).

To start, every vertex is in its own set.



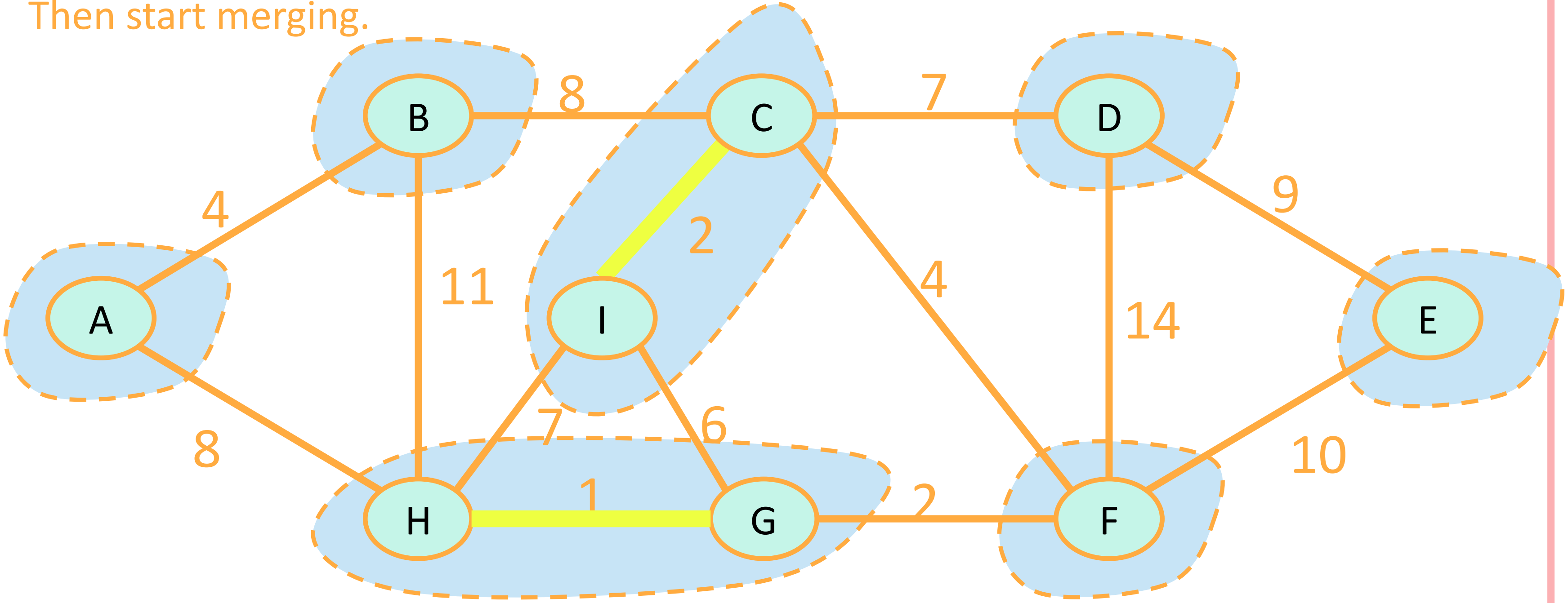
At each step of Kruskal's, we maintain a forest (collection of trees).

Then start merging.



At each step of Kruskal's, we maintain a forest (collection of trees).

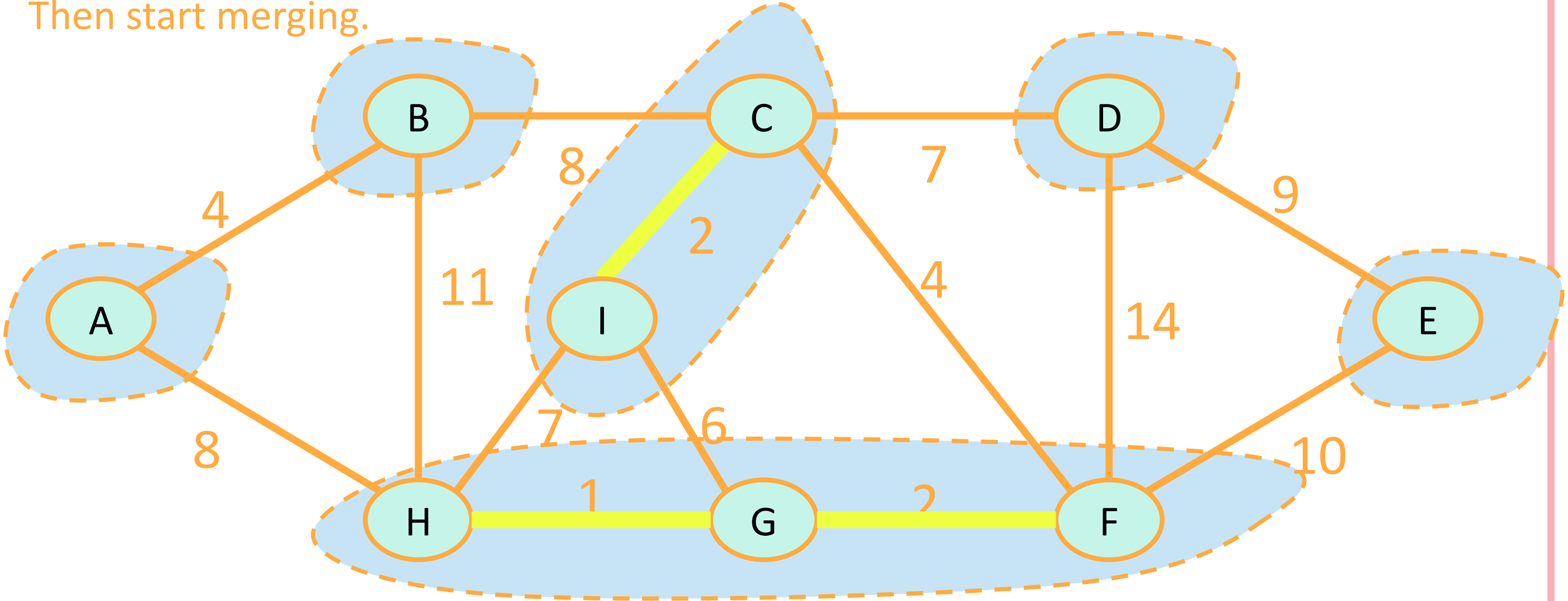
Then start merging.



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

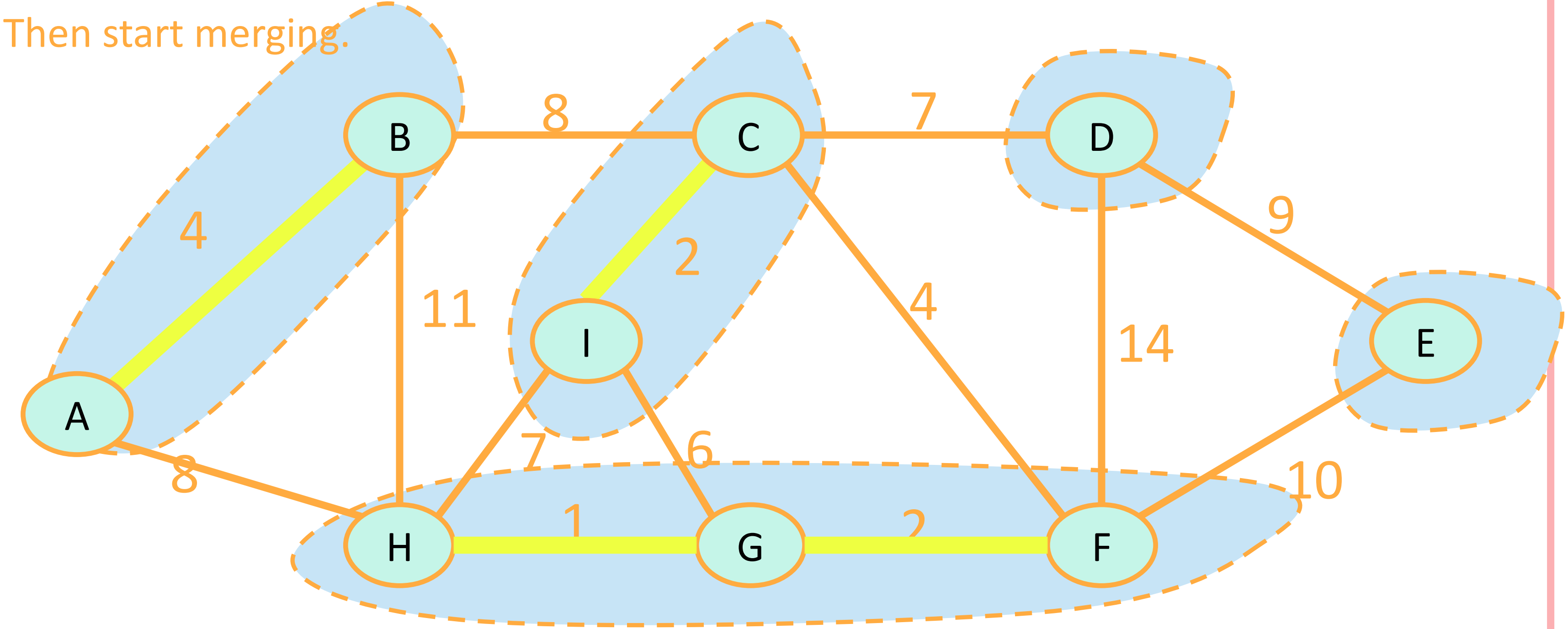
Then start merging.



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

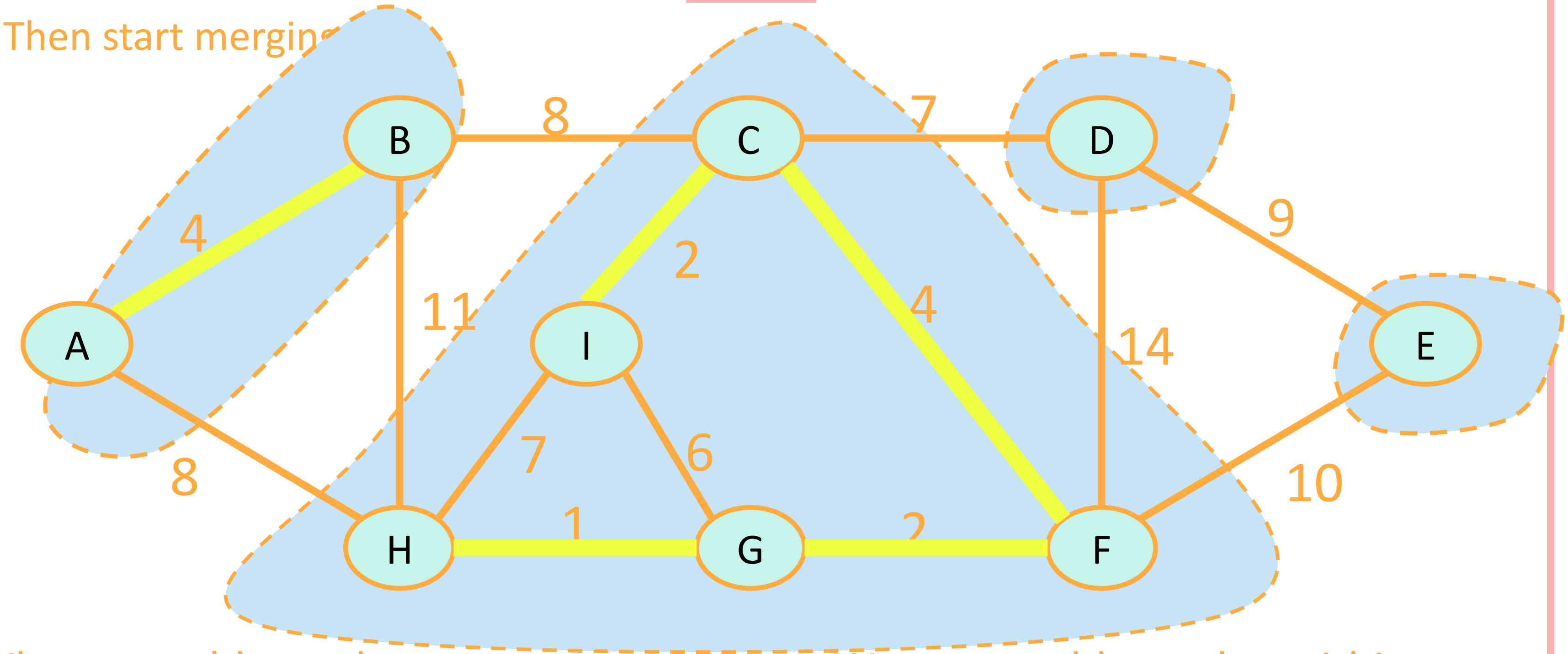
Then start merging.



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

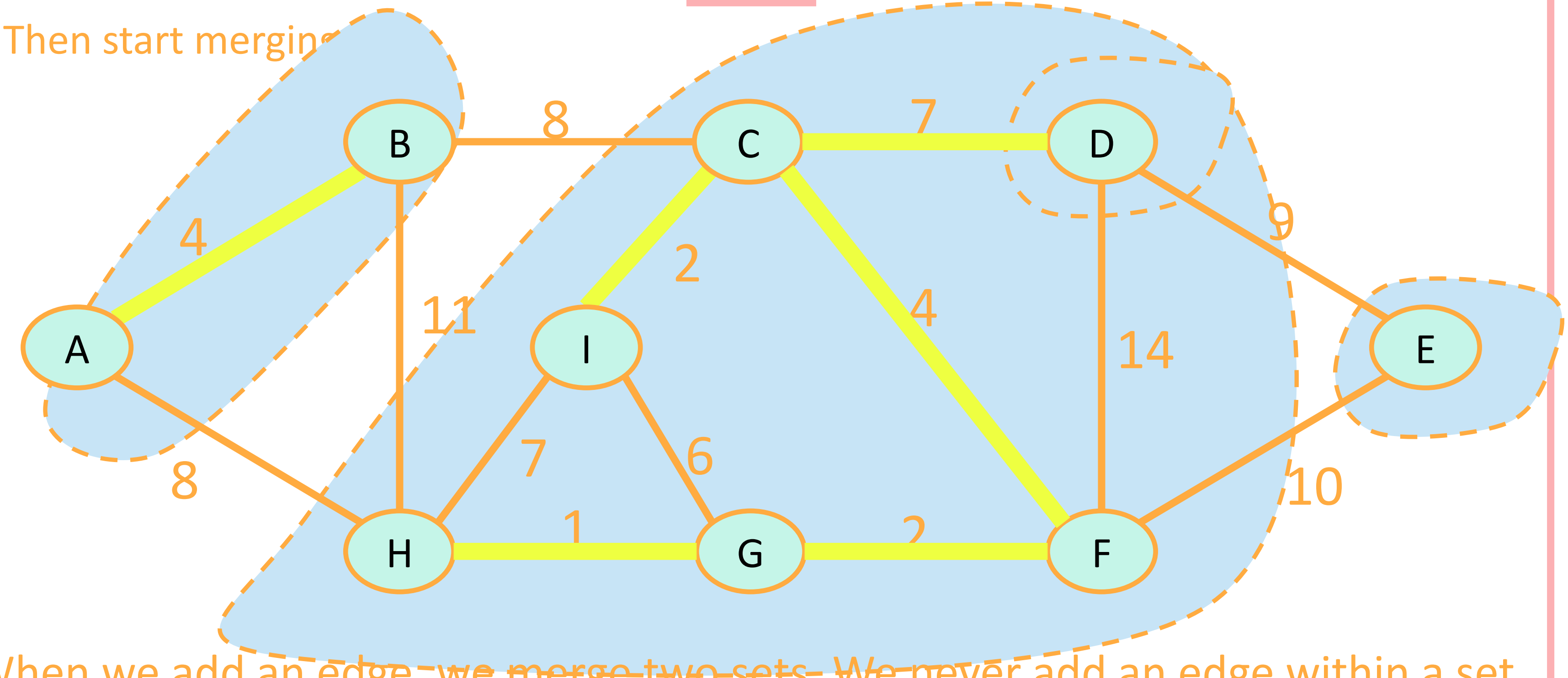
Then start merging



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

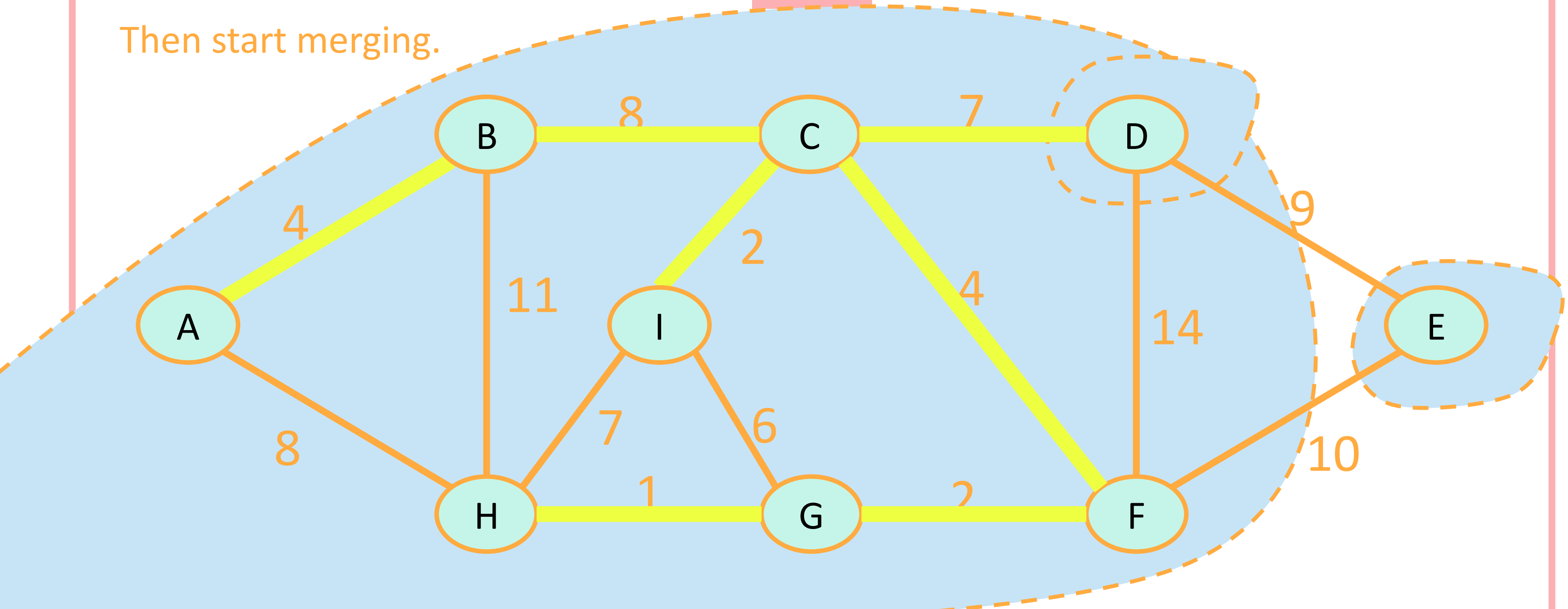
Then start merging



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

Then start merging.

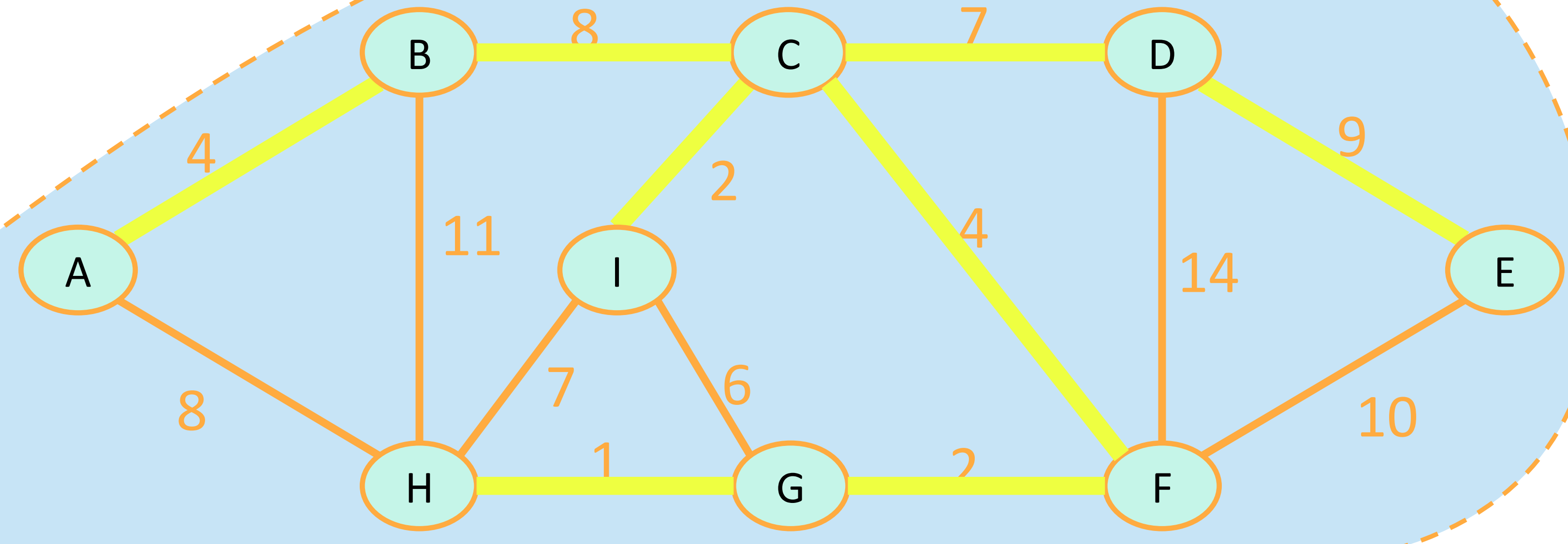


When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

At each step of Kruskal's, we maintain a forest (collection of trees).

Stop when we have one big tree!

Then start merging.



When we add an edge, we merge two sets. We never add an edge within a set since that would create a cycle.

- Worst-case O(Elog(V))

Kruskal Pseudocode (Take 2)

- **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

Big Questions!

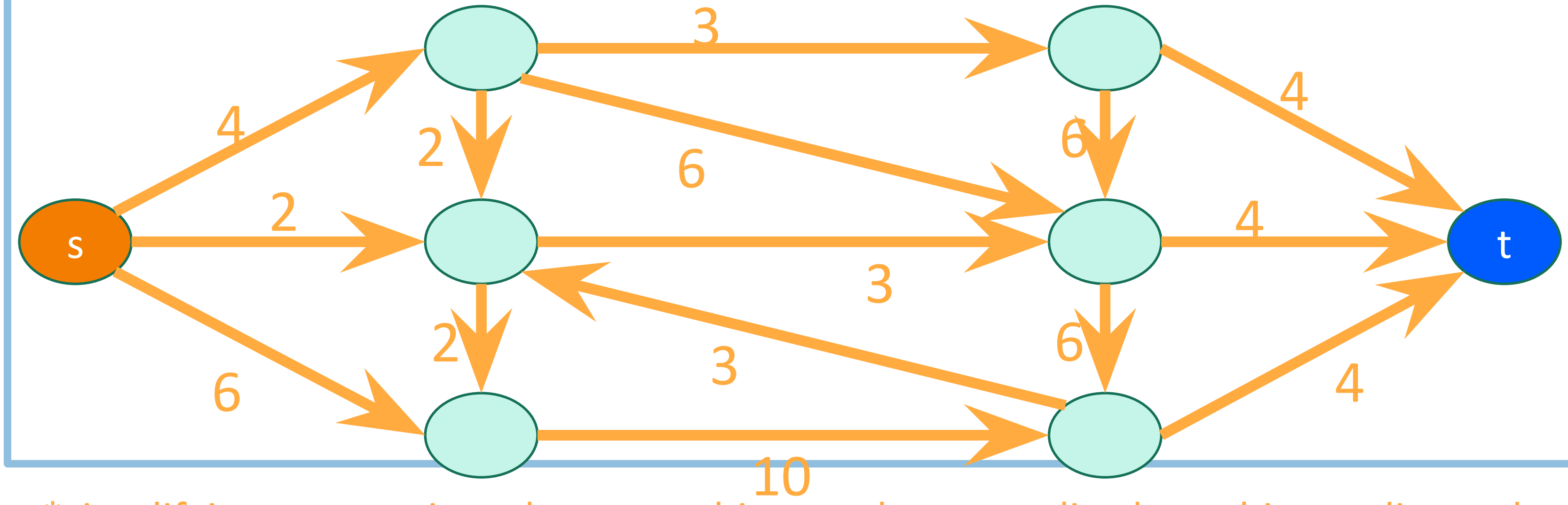
- What about min s-t cuts?
- What are maximum flows?
- How do we find an s-t cut? How do we find max flows?

Big Questions!

- What about min s-t cuts?
- What are maximum flows?
- How do we find an s-t cut? How do we find max flows?

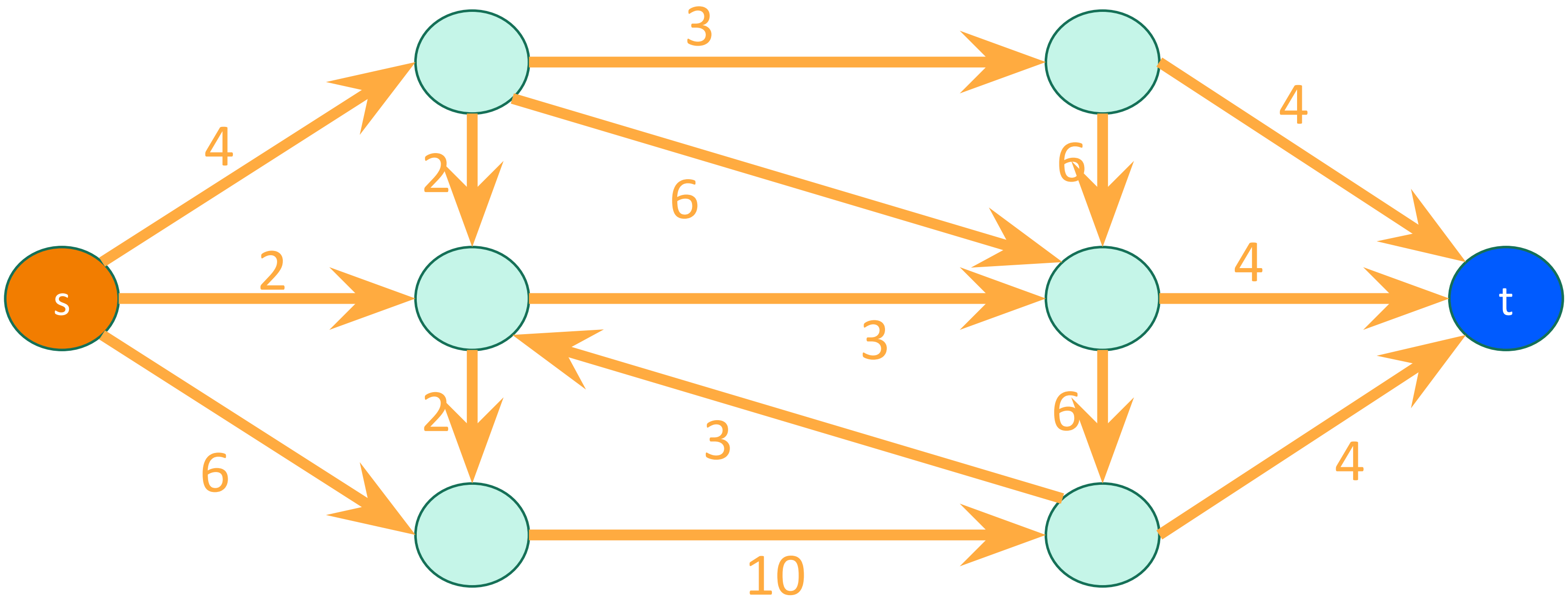
A More General Problem Statement

- 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*

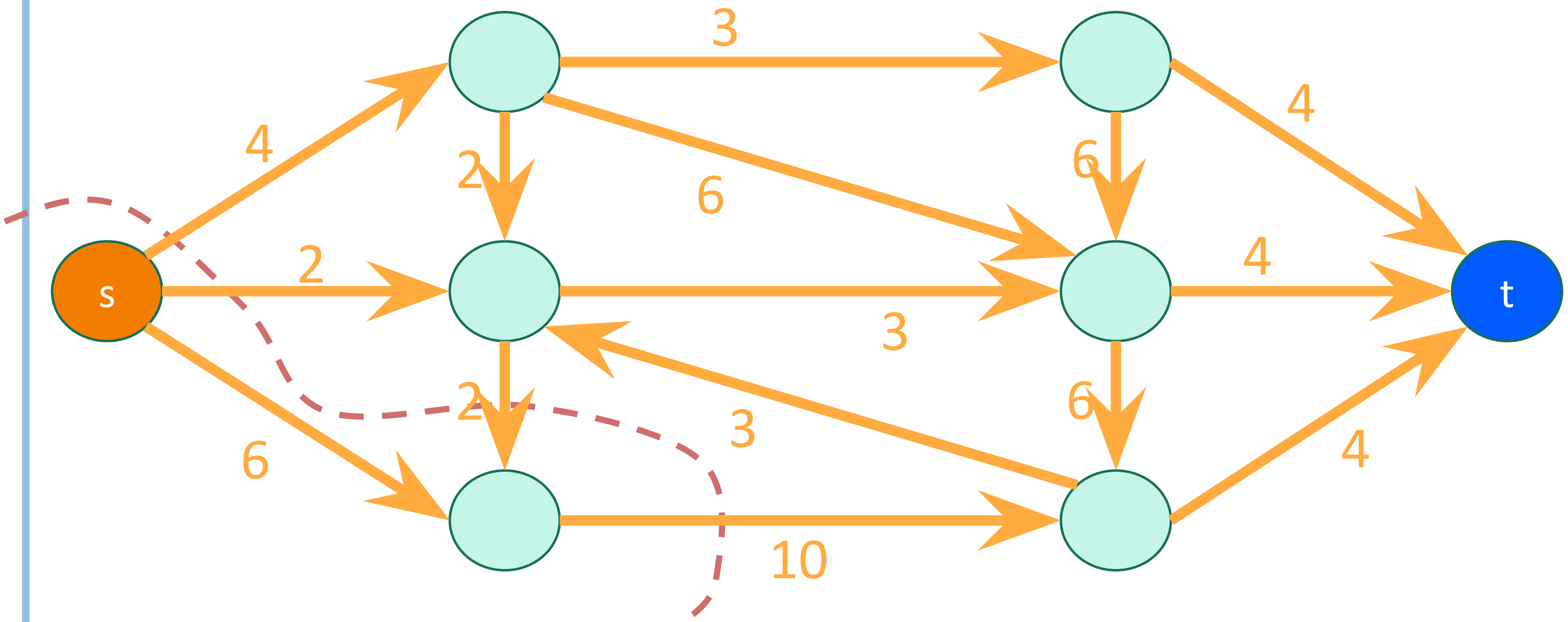


*simplifying assumptions, but everything can be generalized to arbitrary directed graphs

An s-t cut is a cut which separates s from t

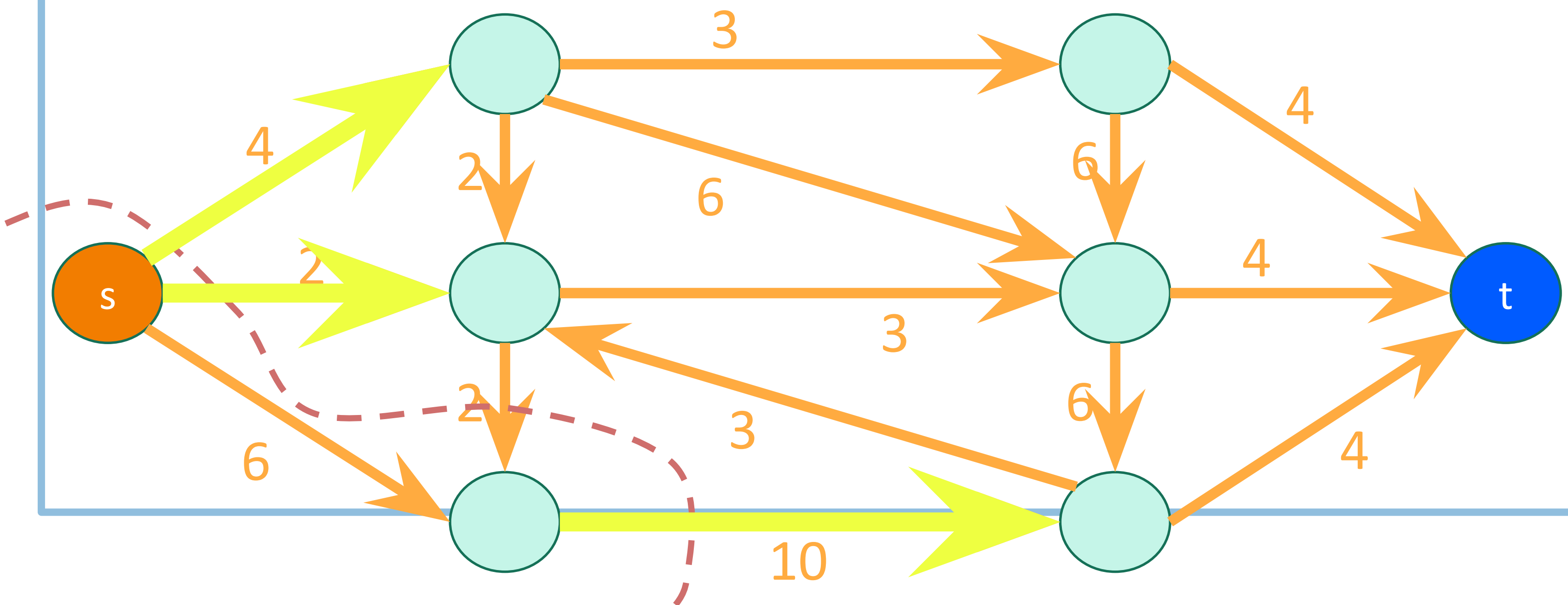


An s-t cut is a cut which separates s from t



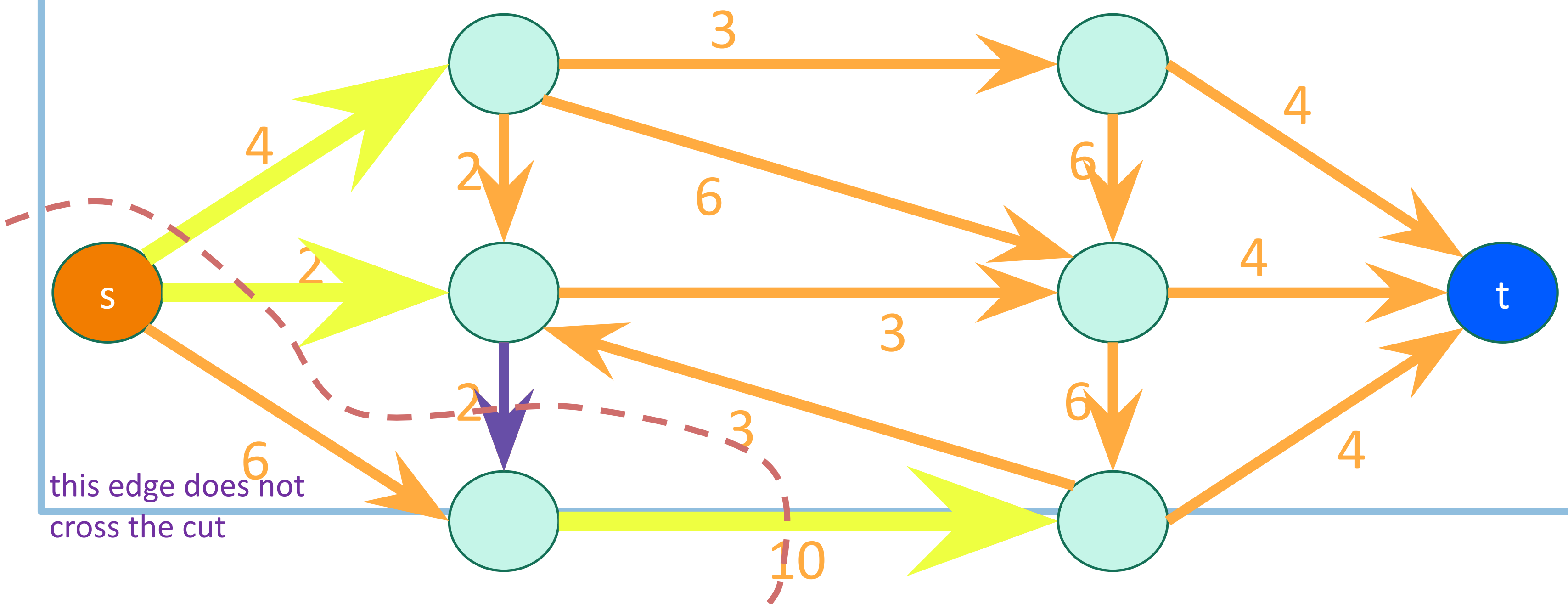
An s-t cut is a cut which separates s from t

- 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.

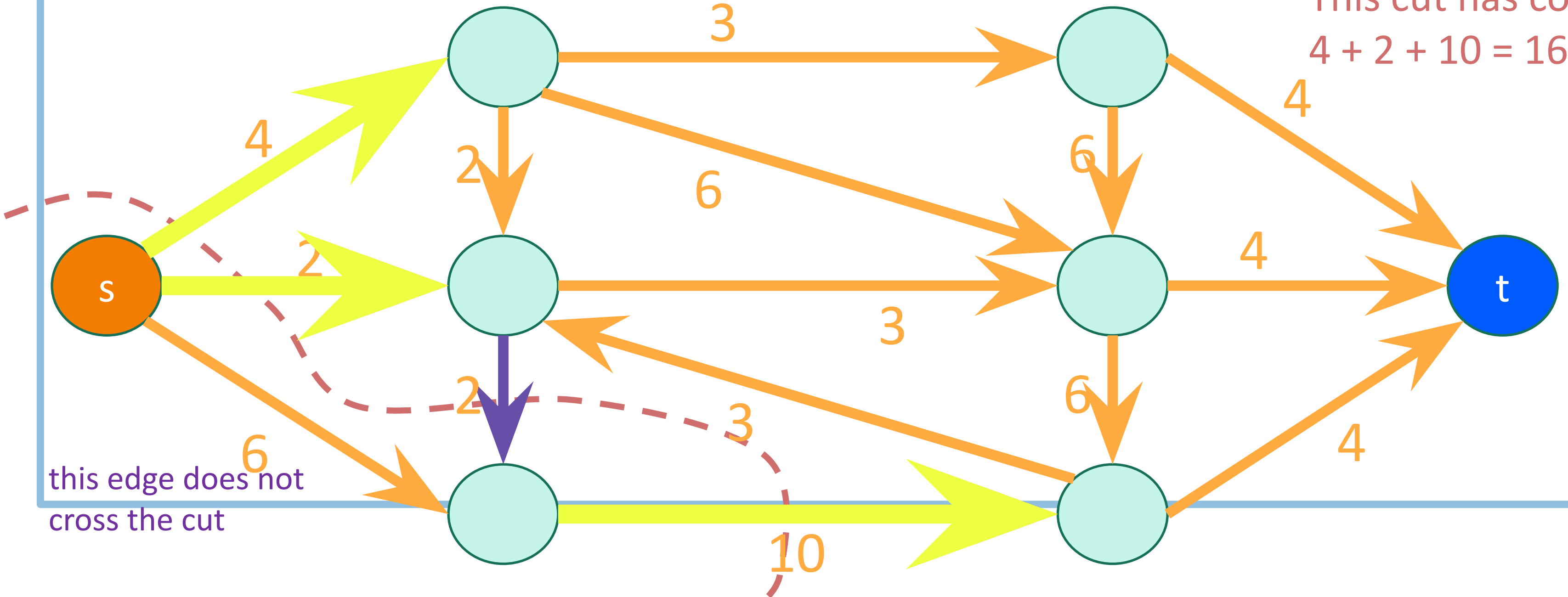


An s-t cut is a cut which separates s from t

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

This cut has cost

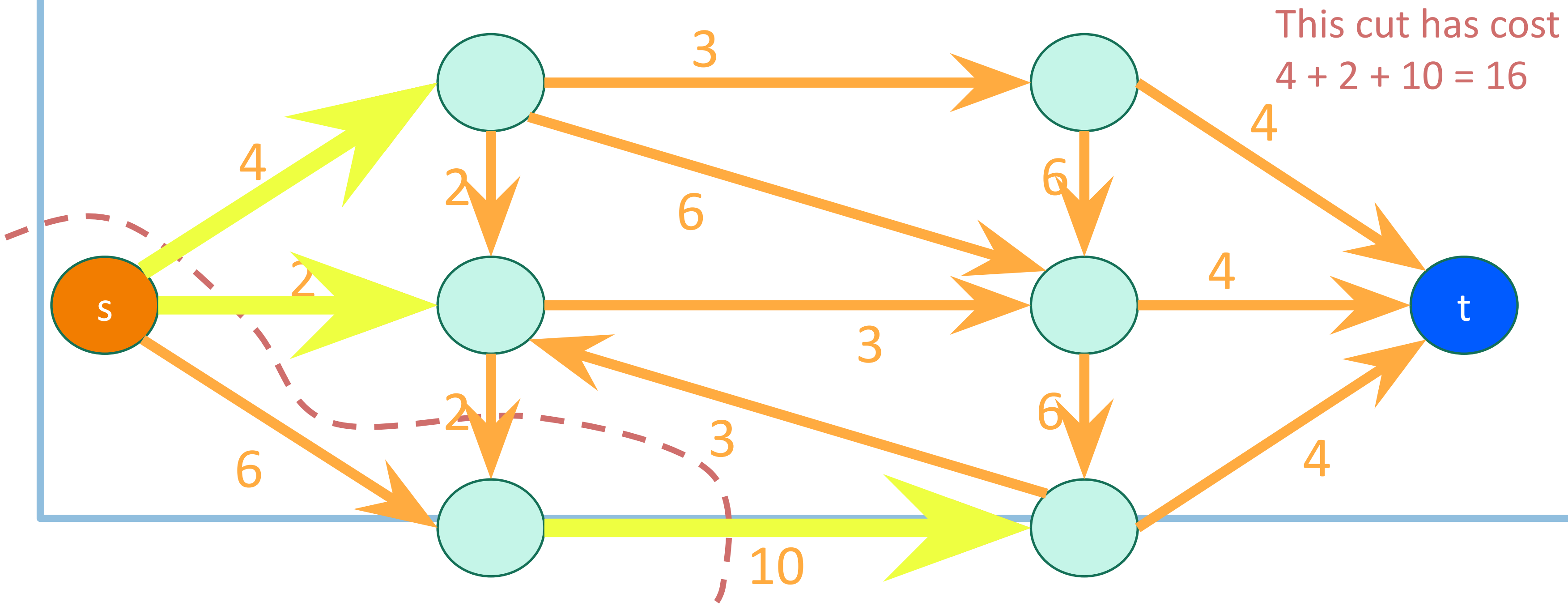
4 + 2 + 10 = 16



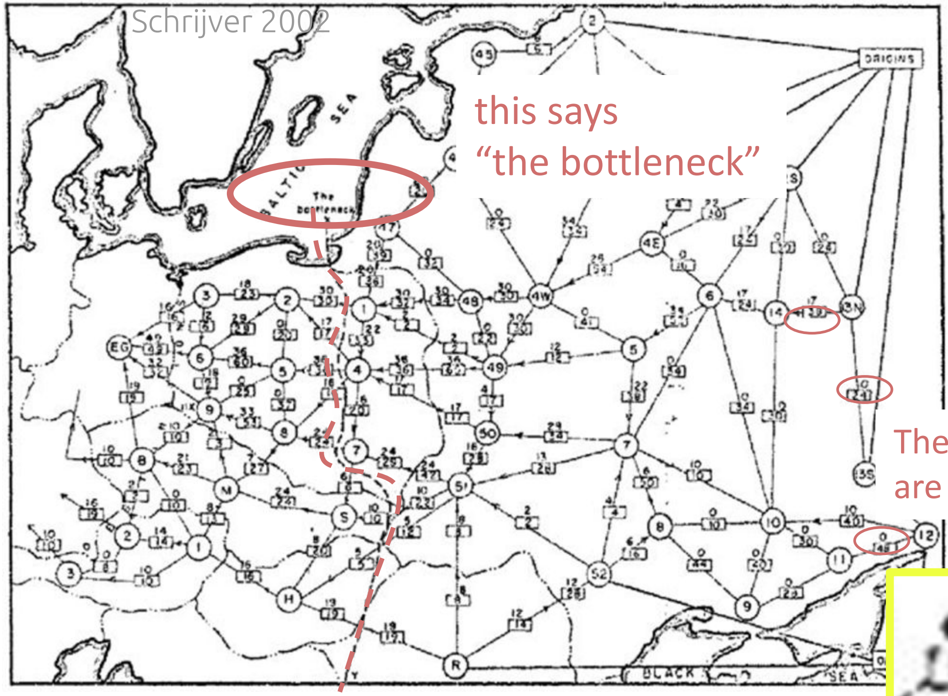
this edge does not cross the cut

A *minimum* s-t cut: a cut which separates s from t with minimum cost.

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



Example where this comes up



this says
“the bottleneck”

These numbers
are capacities.

- 1955 map of rail networks from the Soviet Union to Eastern Europe.
  - Declassified in 1999.
  - 44 edges, 105 vertices

- The US wanted to cut off routes from suppliers in Russia to Eastern Europe as efficiently as possible.

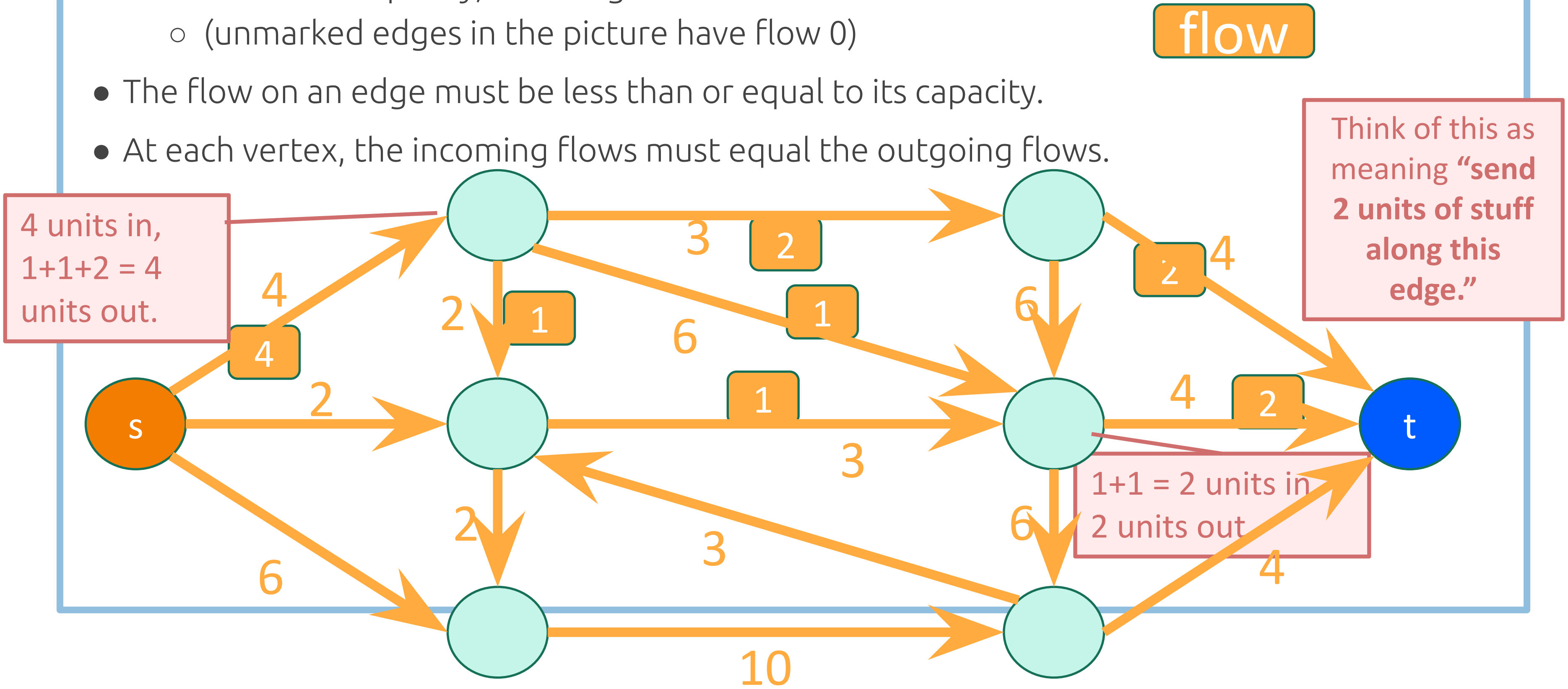
- In 1955, Ford and Fulkerson gave an algorithm which finds the optimal s-t cut.

Big Questions!

- What about min s-t cuts?
- What are maximum flows?
- How do we find an s-t cut? How do we find max flows?

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 or equal to its capacity.
- At each vertex, the incoming flows must equal the outgoing flows.



Flows

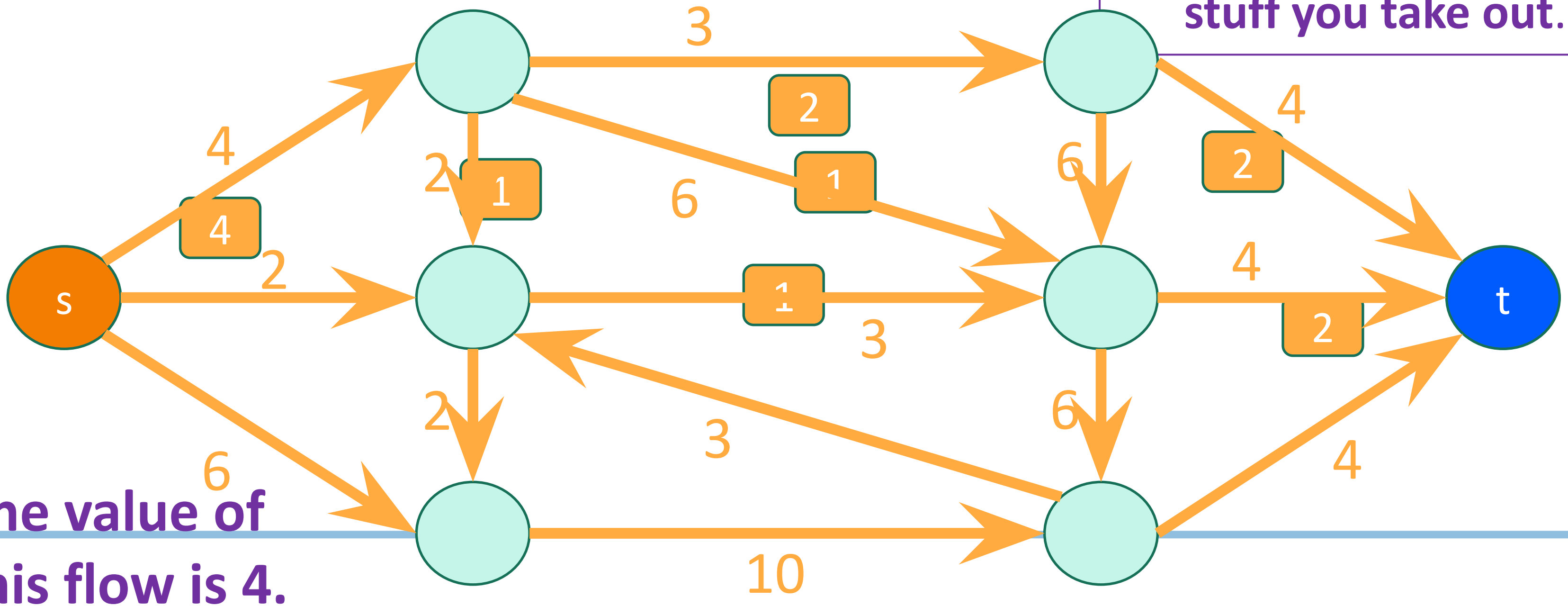
- The value of a flow is:
  - The amount of stuff coming out of s
  - The amount of stuff flowing into t

Because of conservation of flows at vertices,

stuff you put in

=

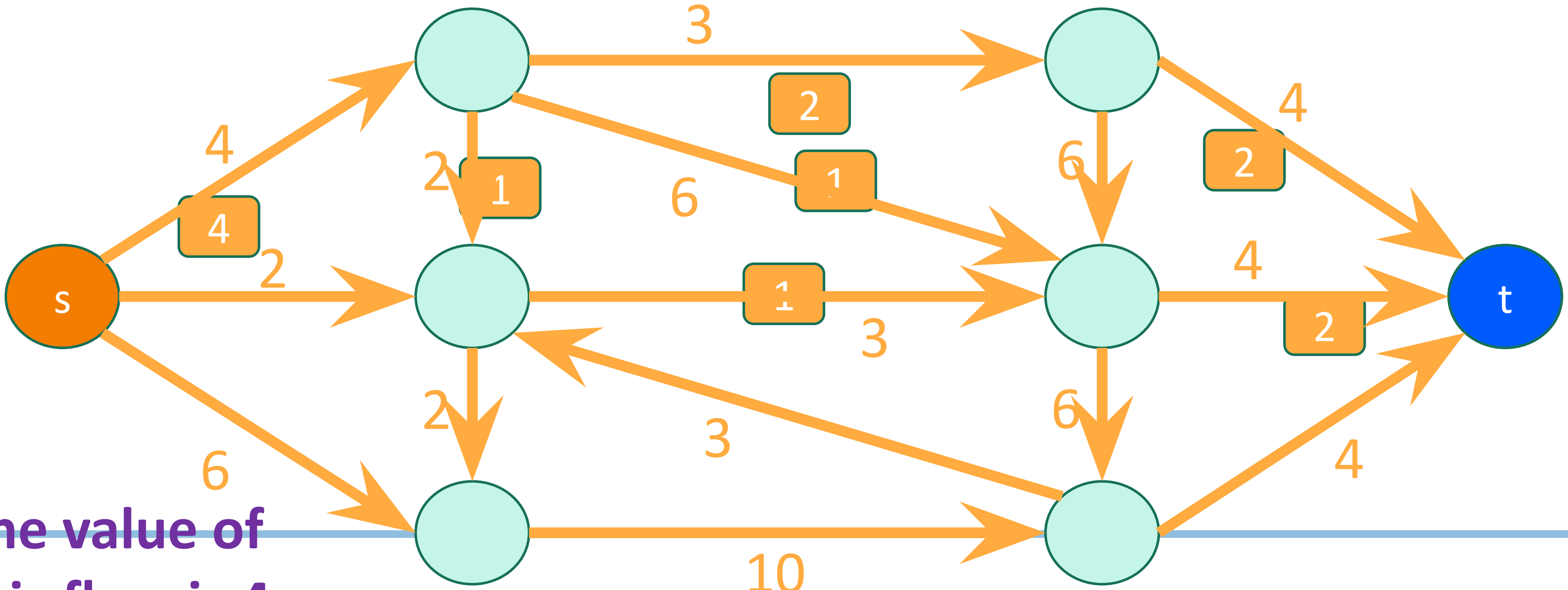
stuff you take out.



The value of this flow is 4.

A *maximum* flow is a flow of maximum value.

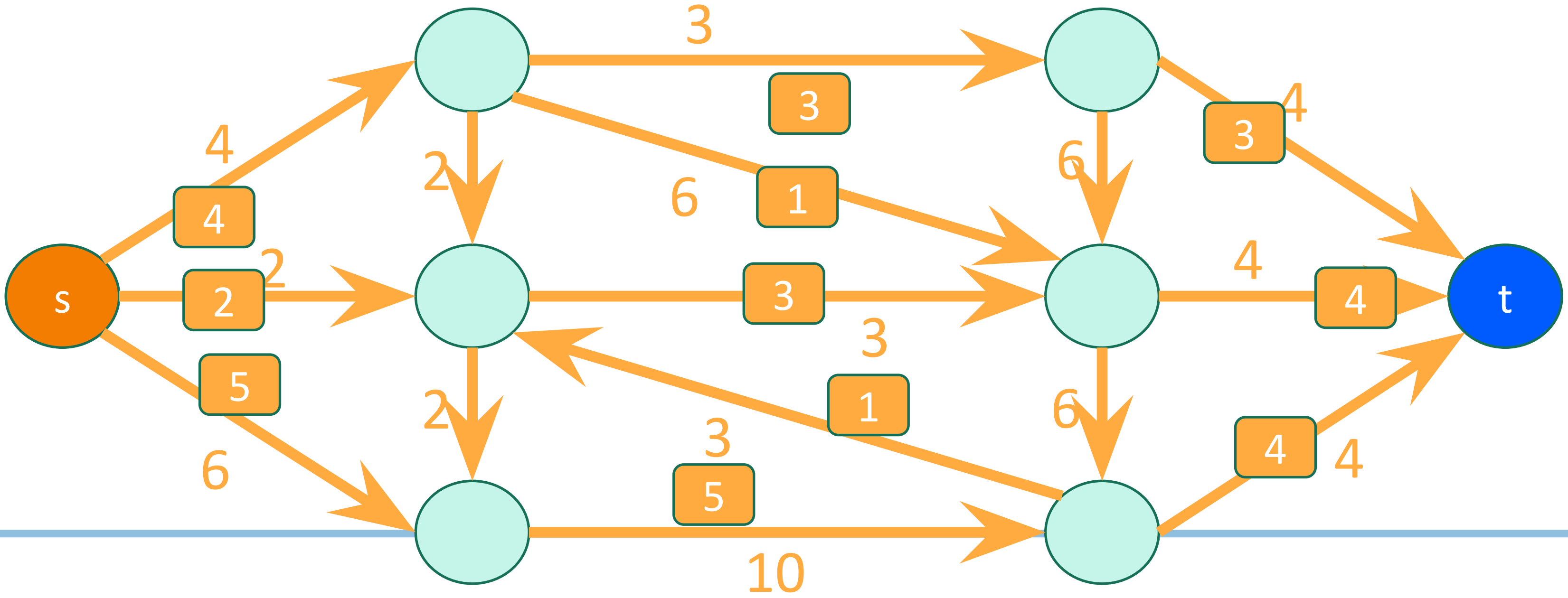
- This example flow is pretty wasteful, I'm not utilizing the capacities very well.



The value of this flow is 4.

A *maximum* flow is a flow of maximum value.

- This one is maximum; it has value 11.



Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Poll

What is the capacity of edge (u, v)?

3

What is the current flow of edge (u, v)?

2

What is the current flow of edge (u, t)?

0

What is the capacity of edge (u, t)?

1

Is vertex u currently following flow conservation?

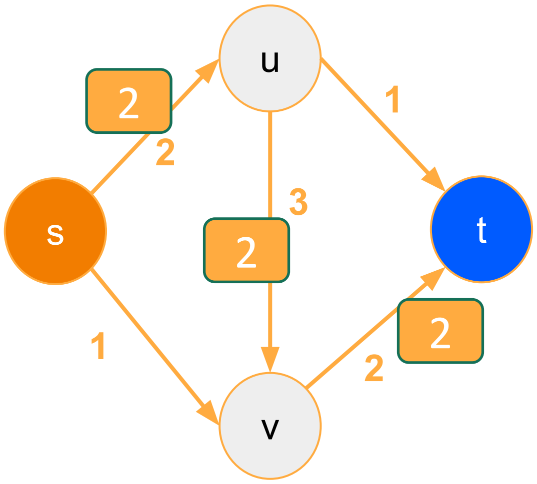
Yes

Is vertex s currently following flow conservation?

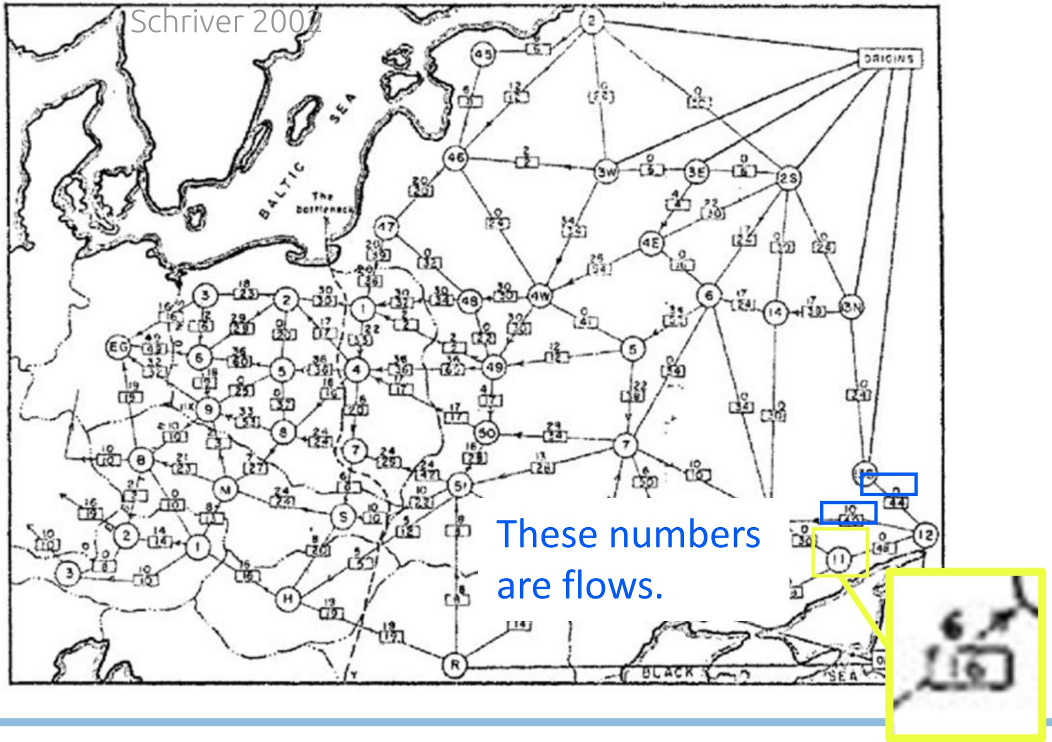
No, but it is not required to

Is this a max flow?

No, the max flow is 3.



Example where this comes up



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

2. Max-Flow Min-Cut

The Soviet rail system also roused the interest of the Americans, and again it inspired fundamental research in optimization.

In their basic paper *Maximal Flow through a Network* (published first as a RAND Report of November 19, 1954), Ford and Fulkerson [5] mention that the maximum flow problem was formulated by T.E. Harris as follows:

> Consider a rail network connecting two cities by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from one given city to the other.

In their 1962 book *Flows in Networks*, Ford and Fulkerson [7] give a more precise reference to the origin of the problem[5]:

> It was posed to the authors in the spring of 1955 by T.E. Harris, who, in conjunction with General F.S. Ross (Ret.), had formulated a simplified model of railway traffic flow, and pinpointed this particular problem as the central one suggested by the model [11].

Ford-Fulkerson's reference 11 is a secret report by Harris and Ross [11] entitled *Fundamentals of a Method for Evaluating Rail Net Capacities*, dated October 24, 1955[6] and written for the US Air Force. At our request, the Pentagon downgraded it to "unclassified" on May 21, 1999.

SECRET

U. S. AIR FORCE

PROJECT RAND

RESEARCH MEMORANDUM

FUNDAMENTALS OF A METHOD FOR EVALUATING RAIL NET CAPACITIES (U)

T. E. Harris
F. S. Ross

RM-1573

October 24, 1955

Copy No. 37

SUMMARY

Air power is an effective means of interdicting an enemy's rail system, and such usage is a logical and important mission for this Arm.

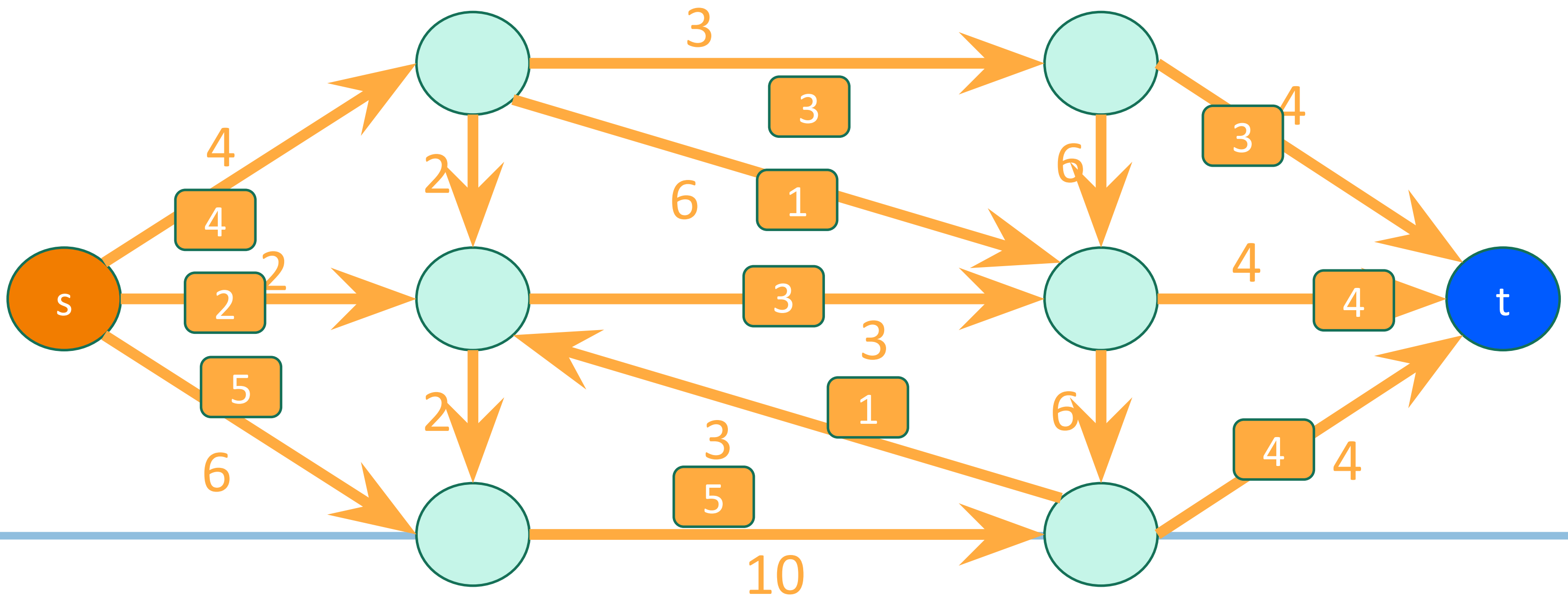
As in many military operations, however, the success of interdiction depends largely on how complete, accurate, and timely is the commander's information, particularly concerning the effect of his interdiction-program efforts on the enemy's capability to move men and supplies. This information should be available at the time the results are being achieved.

The present paper describes the fundamentals of a method intended to help the specialist who is engaged in estimating railway capacities, so that he might more readily accomplish this purpose and thus assist the commander and his staff with greater efficiency than is possible at present.

Theorem: Max-flow min-cut theorem

- This one is maximum; it has value 11.

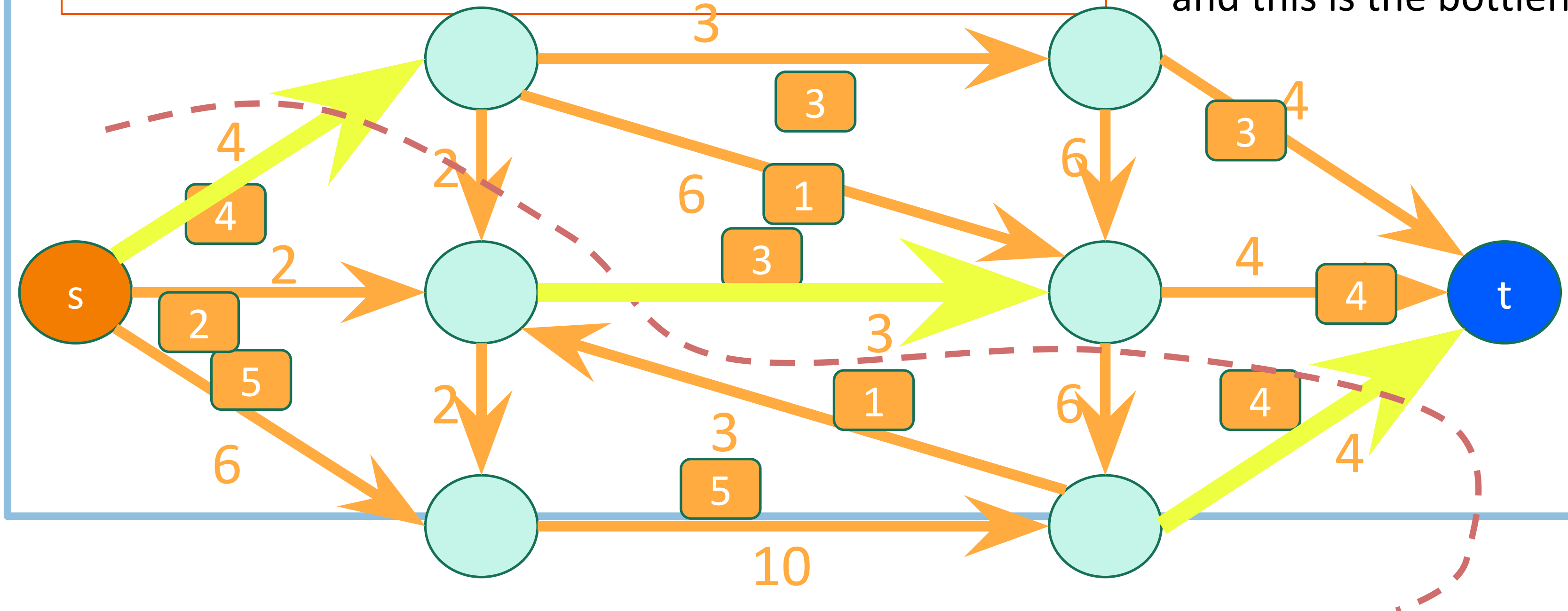
What's the min s-t cut?



Theorem: Max-flow min-cut theorem

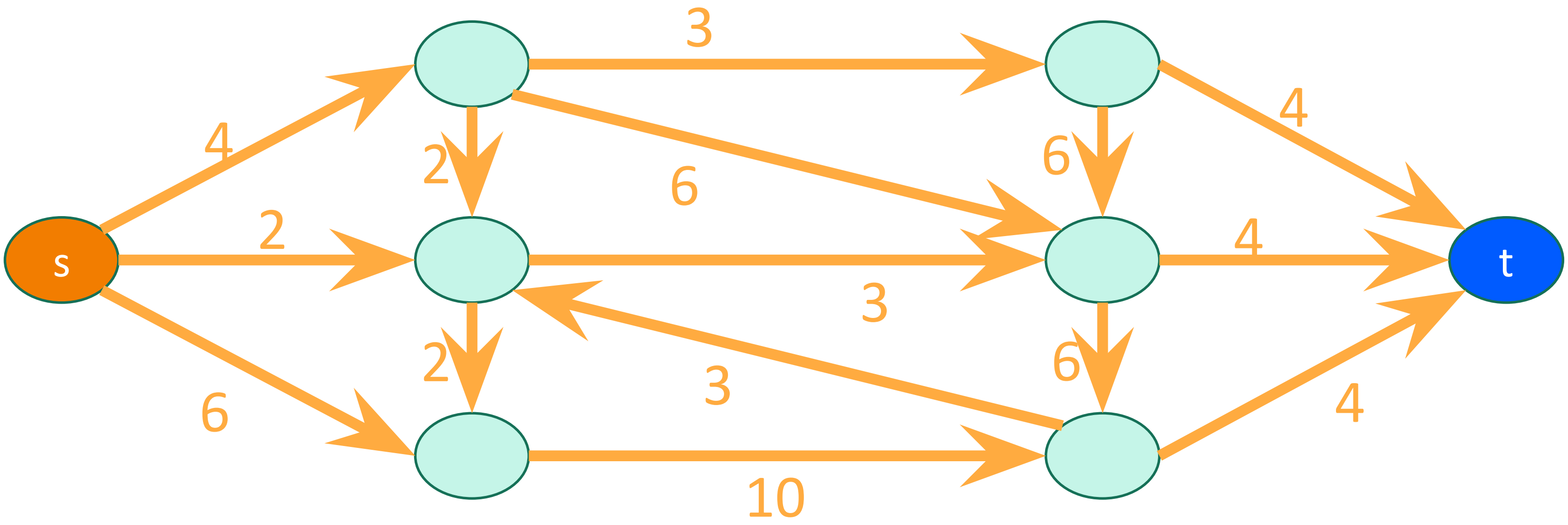
The value of a max flow from s to t
is equal to
the cost of a min s-t cut.

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



Maximum flow

- Brainstorm some algorithms for maximum flow…

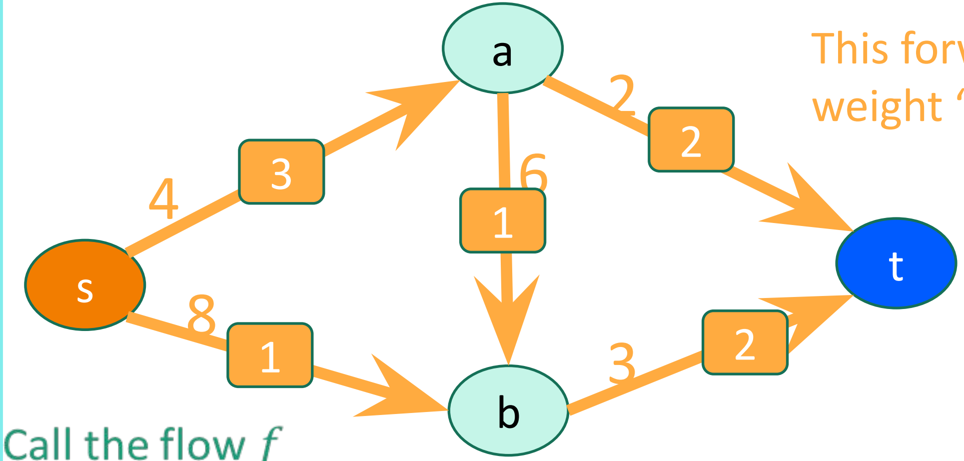


Big Questions!

- What about min s-t cuts?
- What are maximum flows?
- How do we find an s-t cut? How do we find max flows?

Residual networks

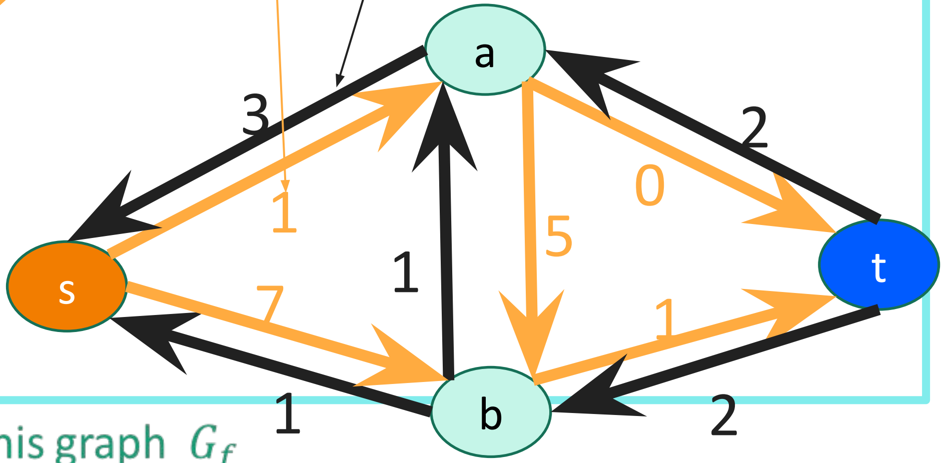
Say we have a flow…



Call the flow f
Call the graph G

This forward edge has weight "capacity – flow".

This backward edge has weight "flow".

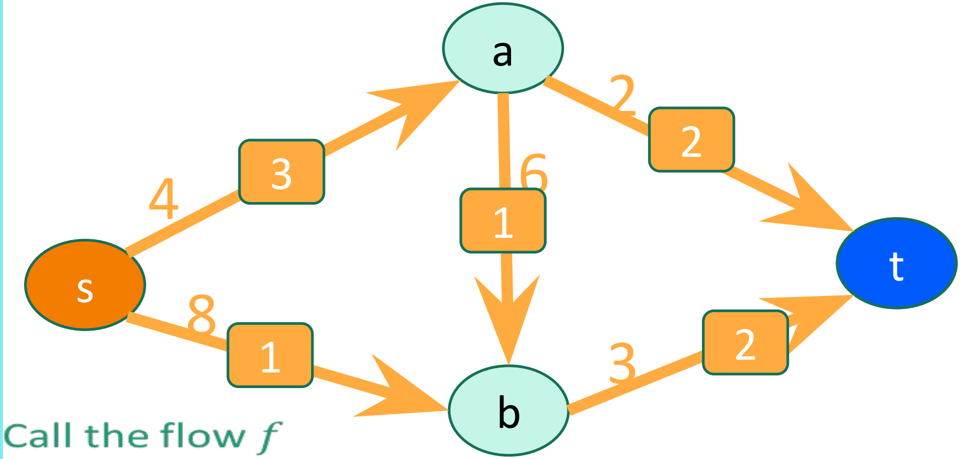


Create a new **residual network** from this flow:

Call this graph G_f

Residual networks

Say we have a flow…

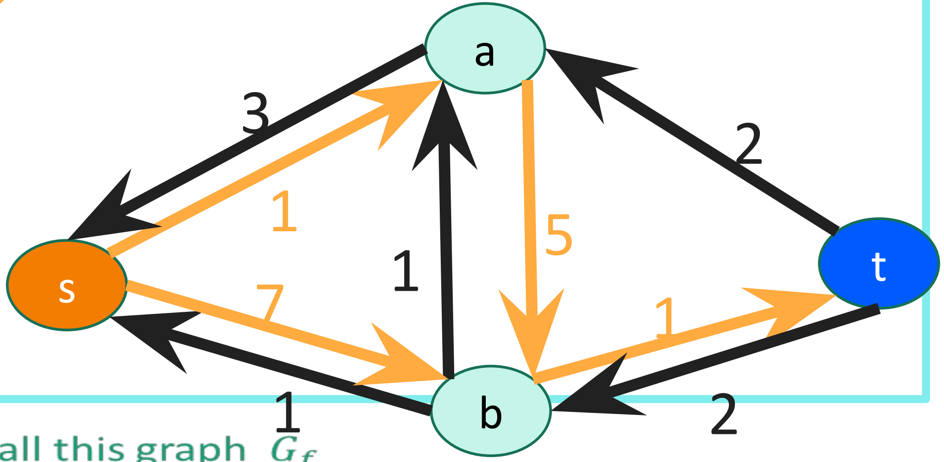


Call the flow f
Call the graph G

Forward edges are the amount that's left.

Backwards edges are the amount that's been used.

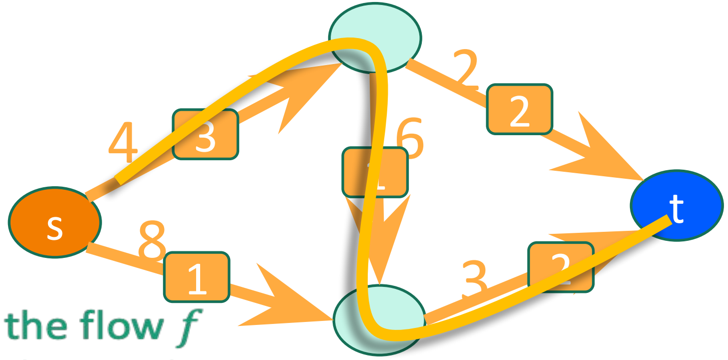
Create a new **residual network** from this flow:



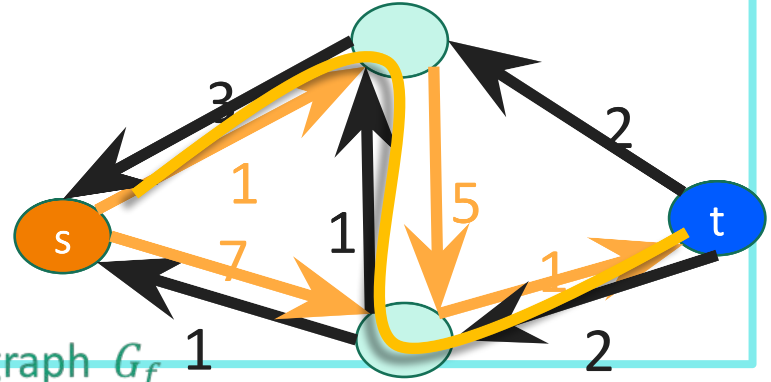
Call this graph G_f

Residual networks tell us how to improve the flow.

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



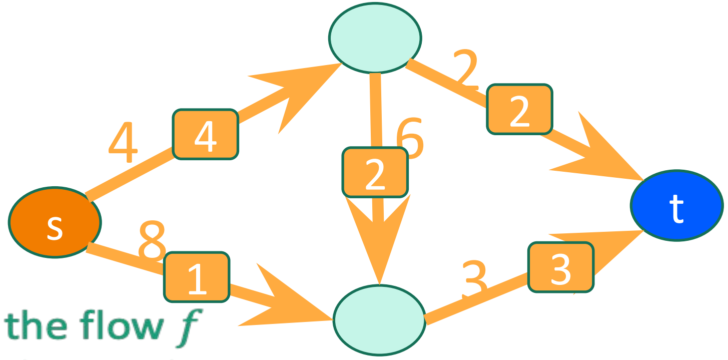
Call the flow f
Call the graph G



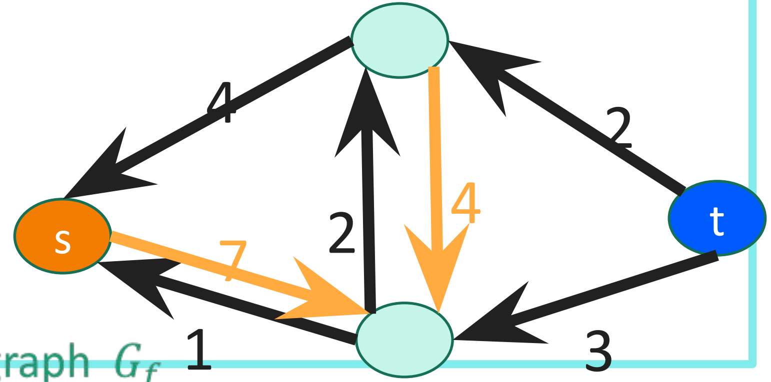
Call this graph G_f

Residual networks tell us how to improve the flow.

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



Call the flow f
Call the graph G

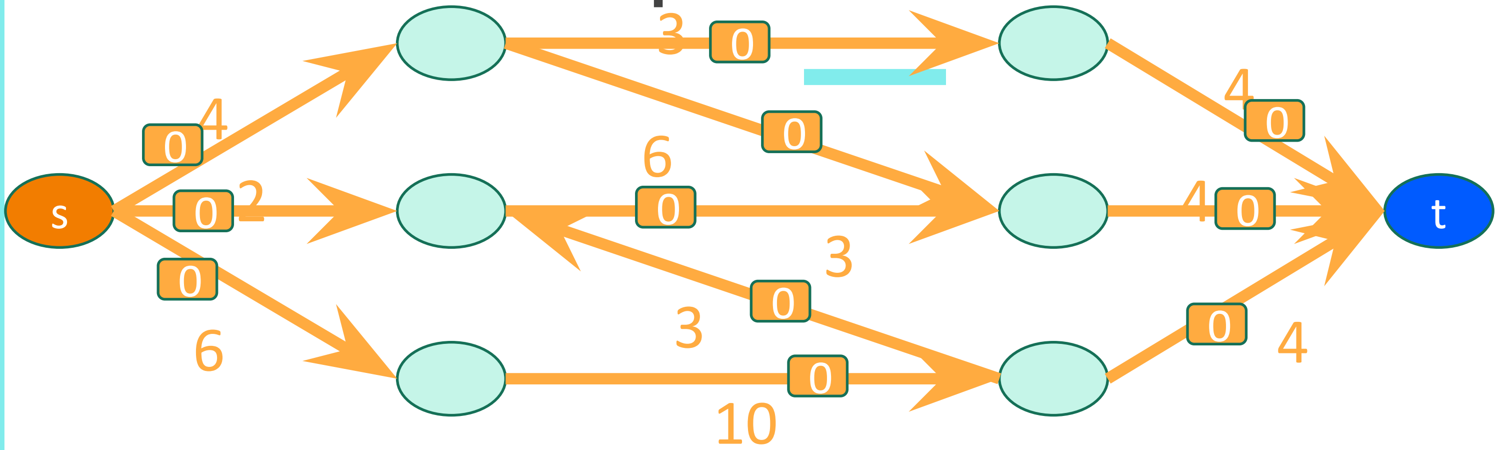


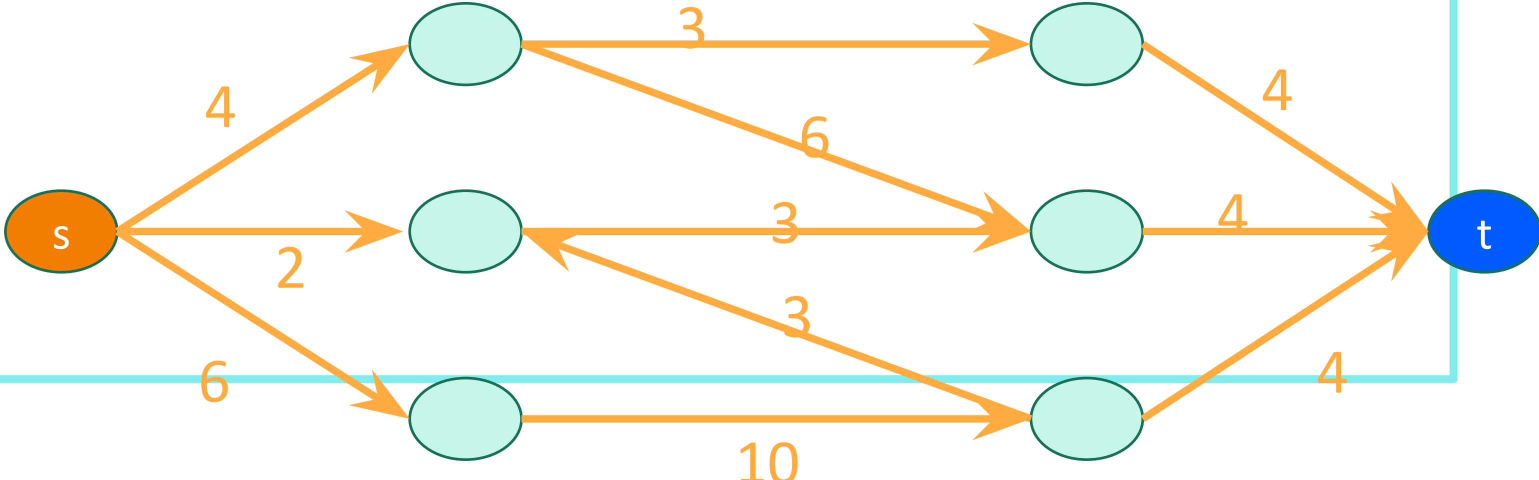
Call this graph G_f

Ford-Fulkerson Algorithm

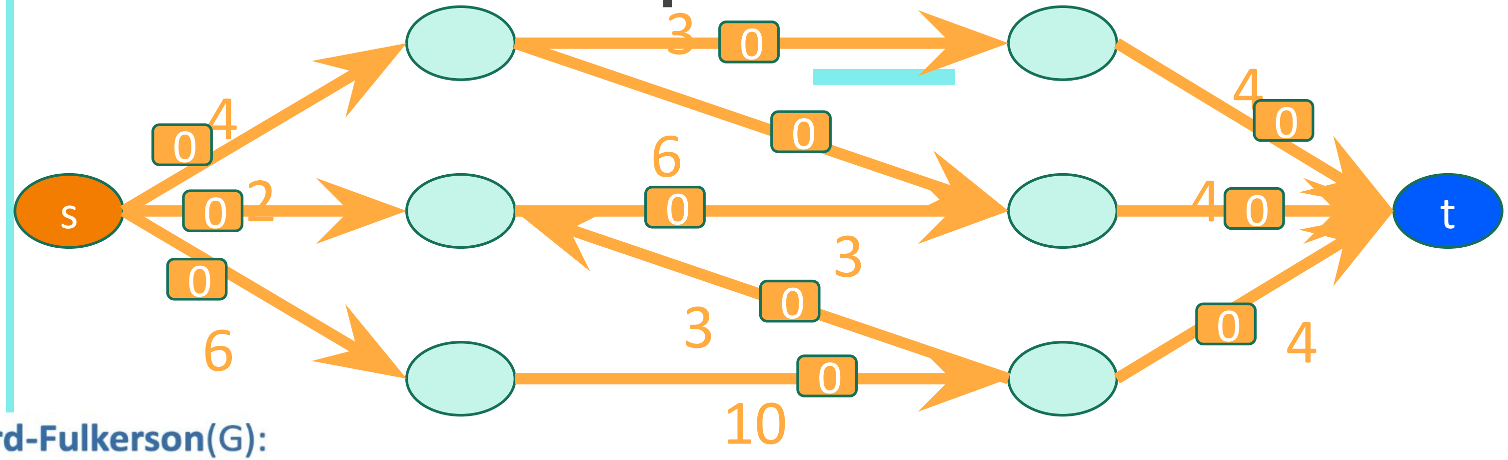
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f // e.g., use DFS or BFS
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f

Example of Ford-Fulkerson

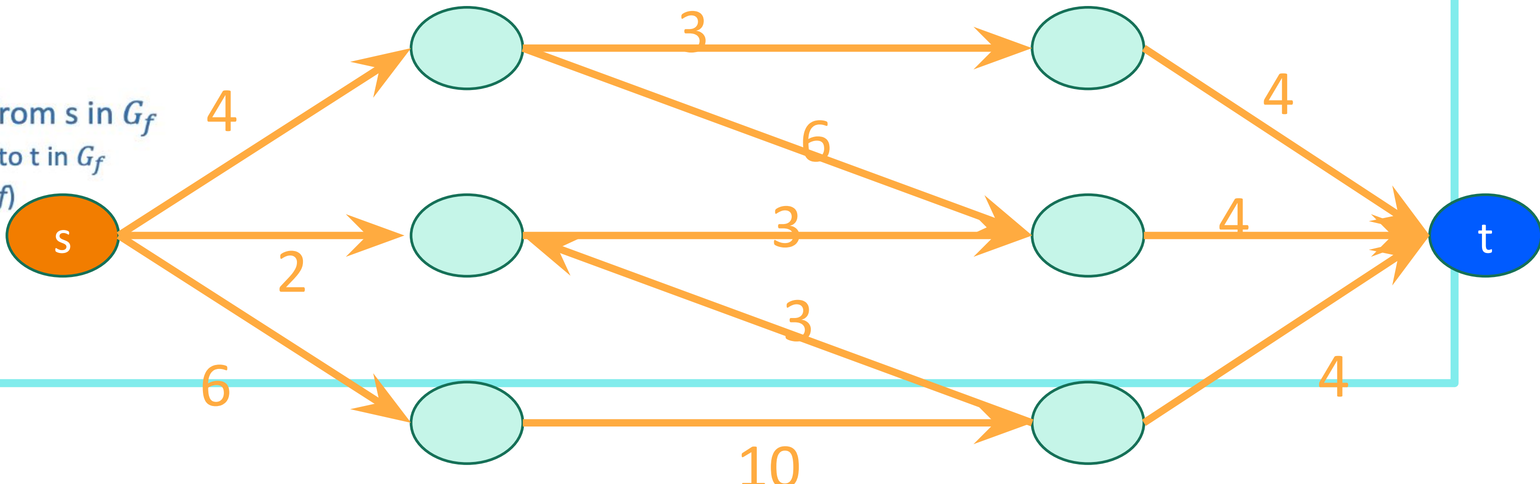




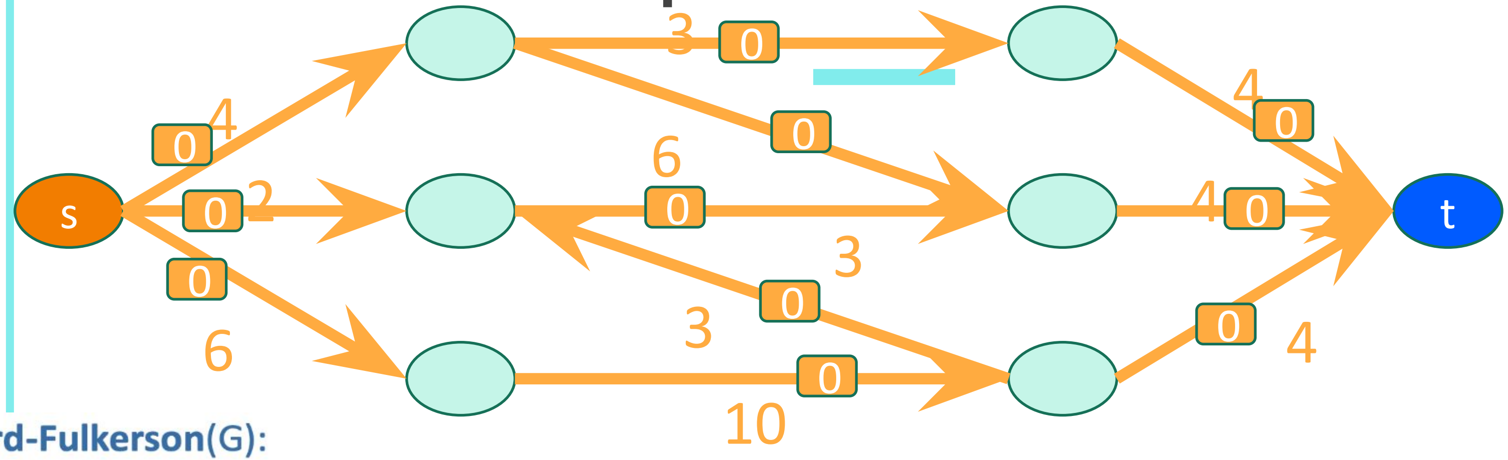
Example of Ford-Fulkerson



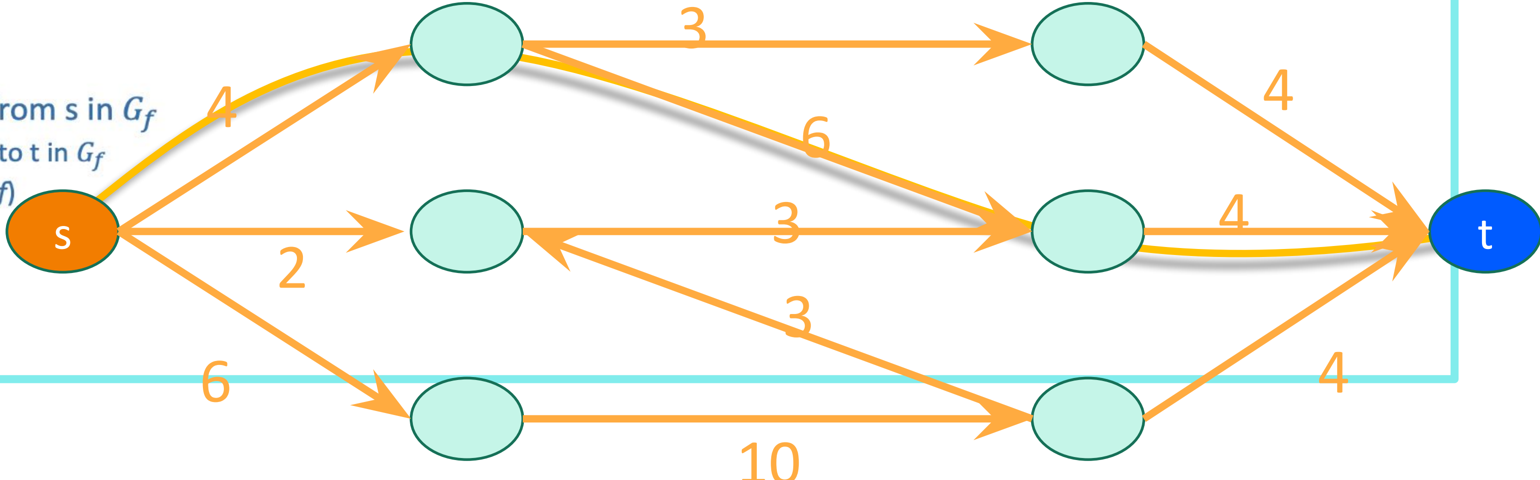
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



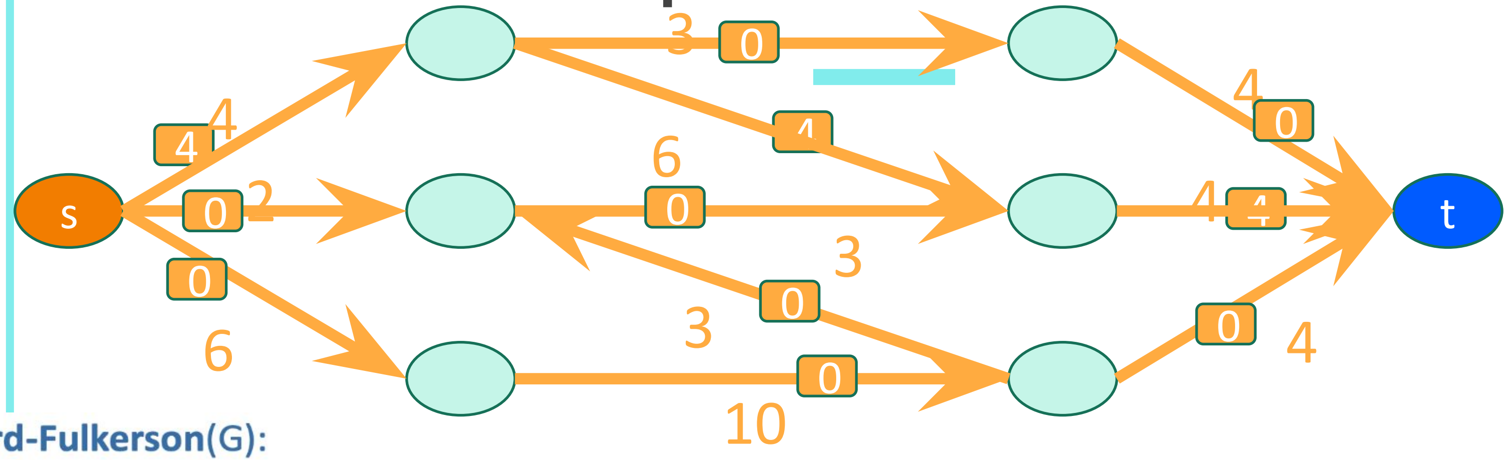
Example of Ford-Fulkerson



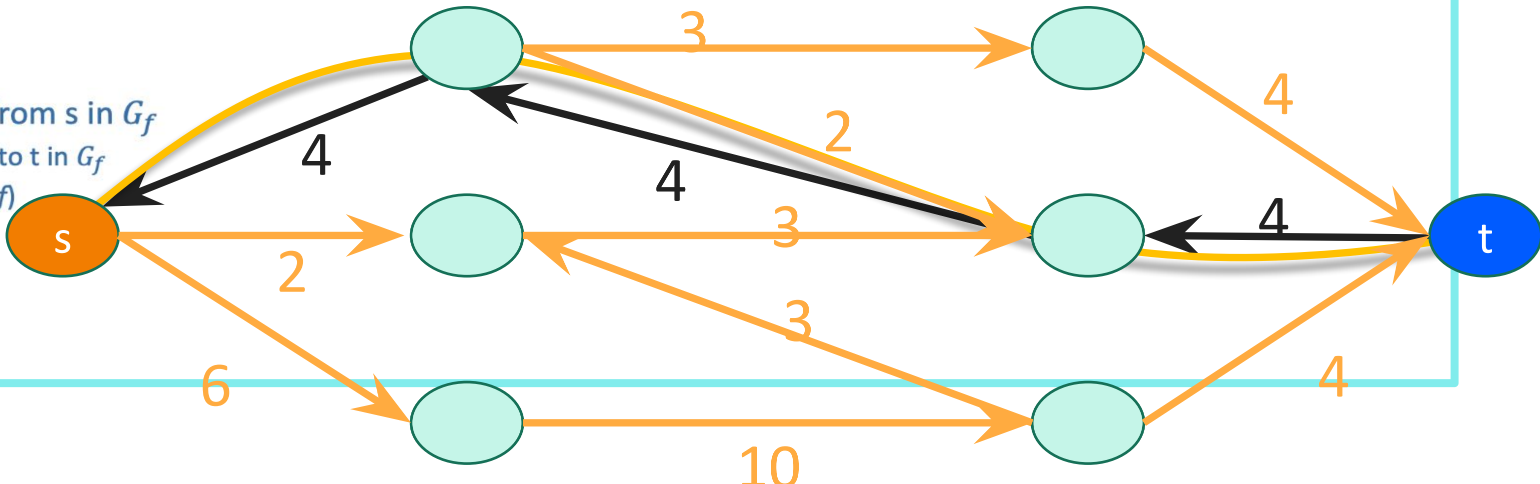
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



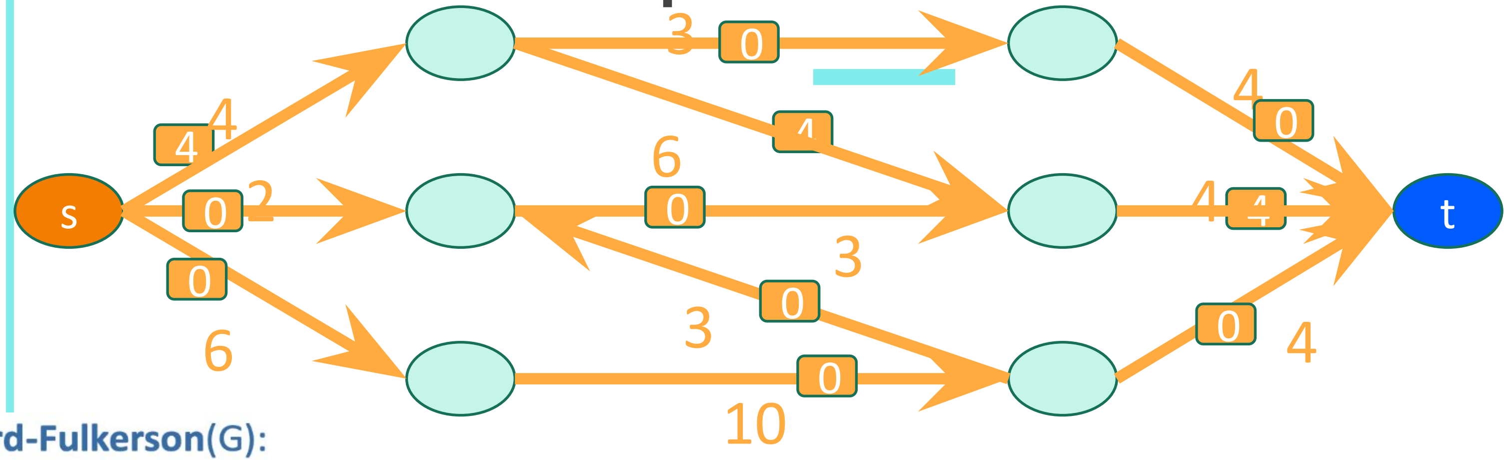
Example of Ford-Fulkerson



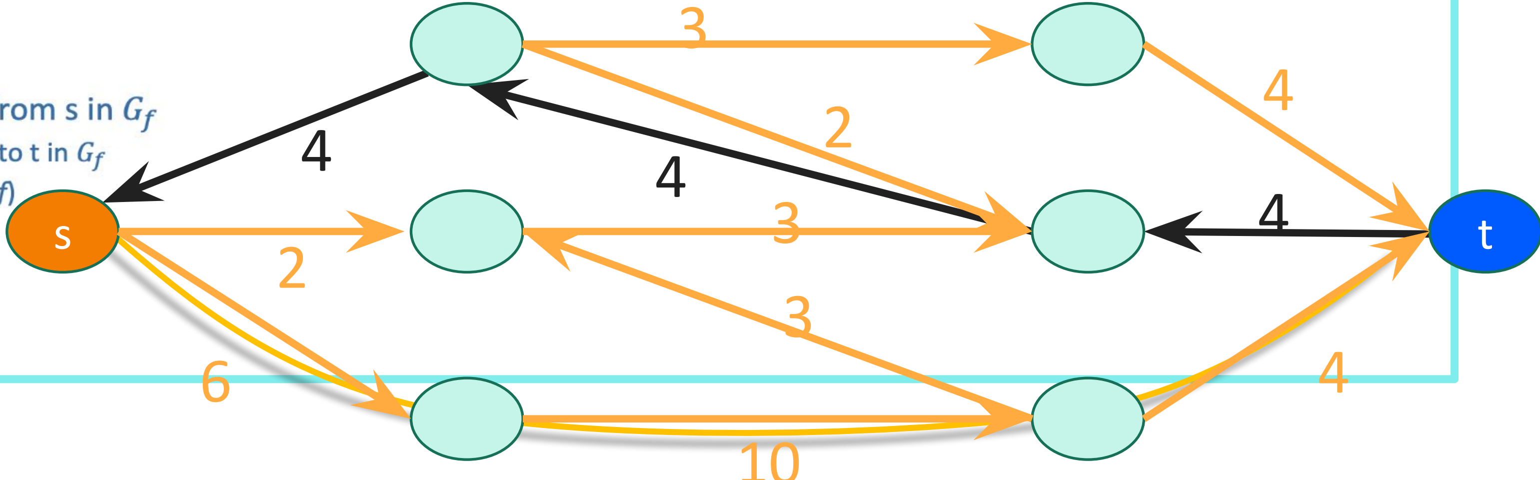
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



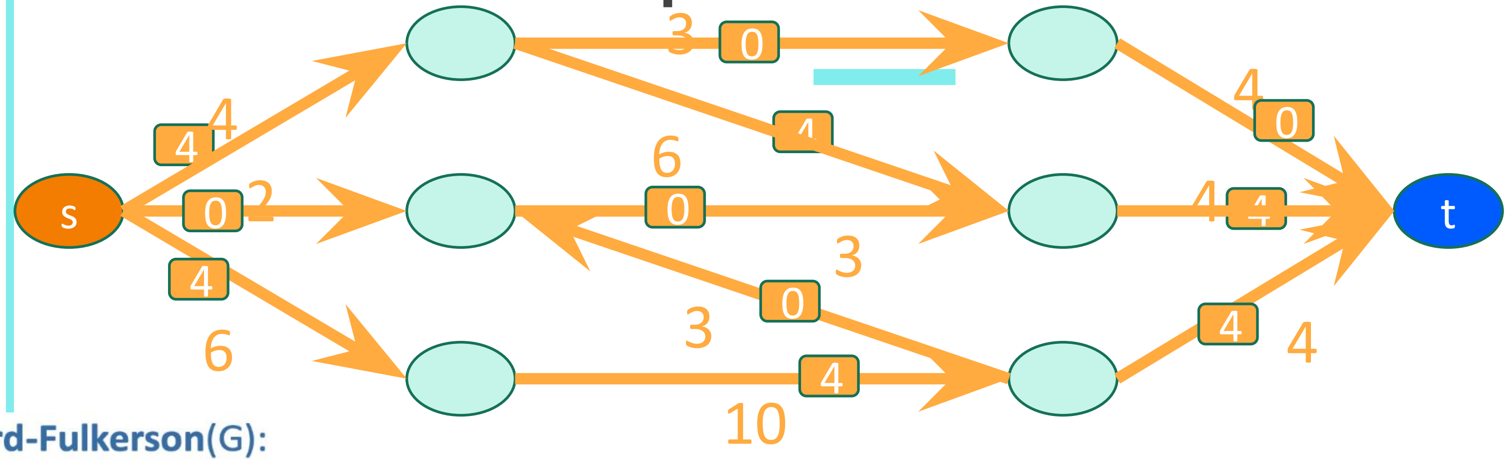
Example of Ford-Fulkerson



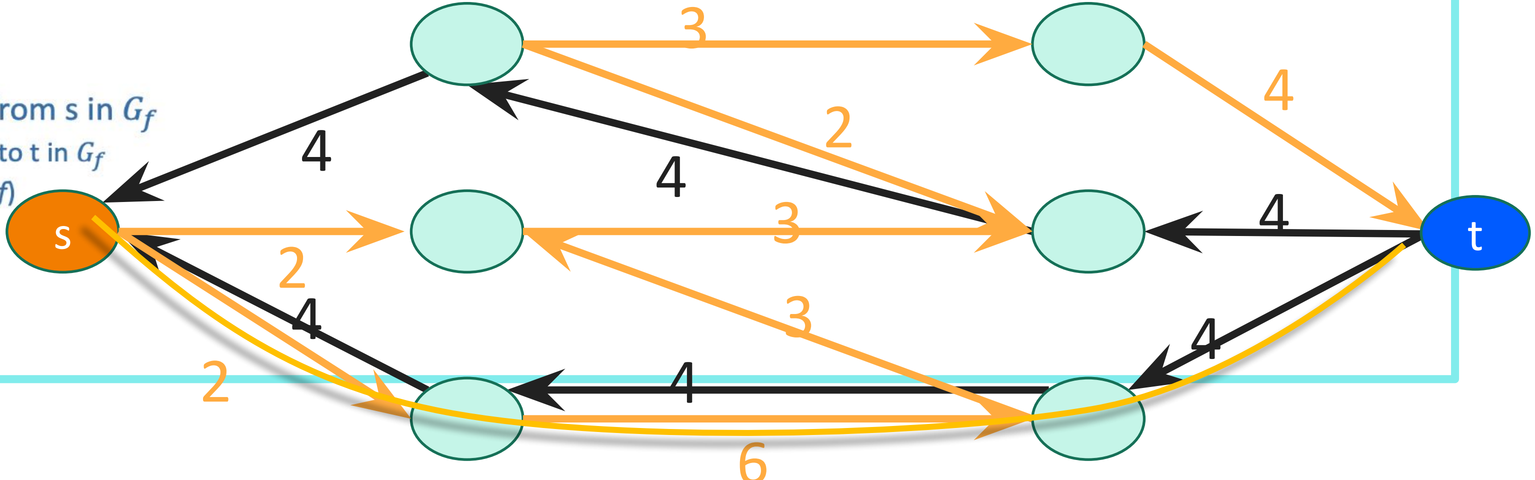
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



Example of Ford-Fulkerson

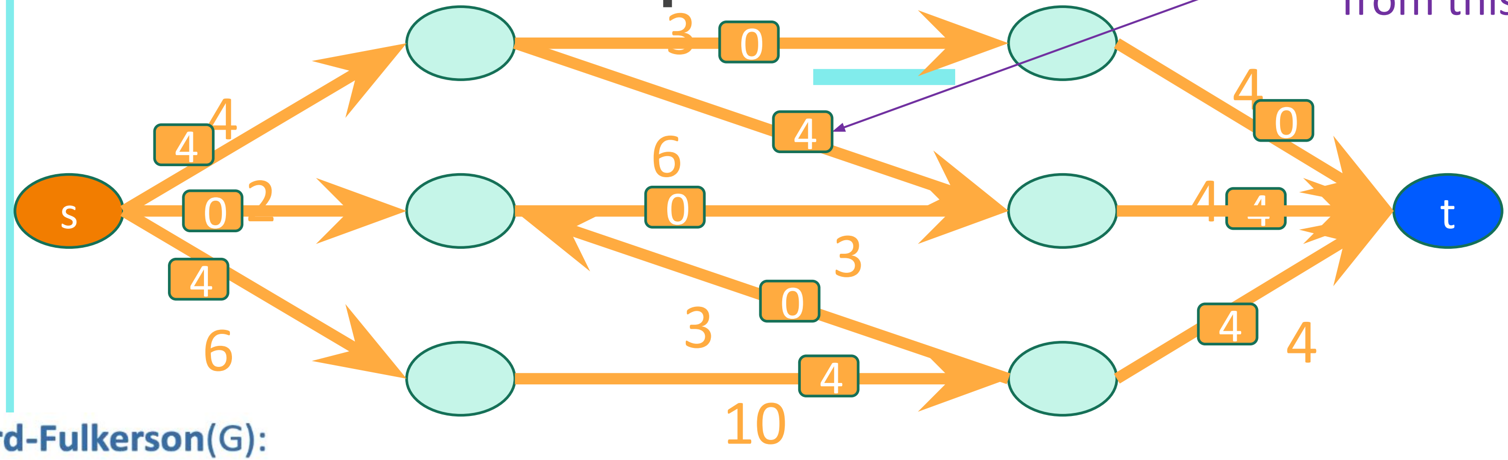


- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



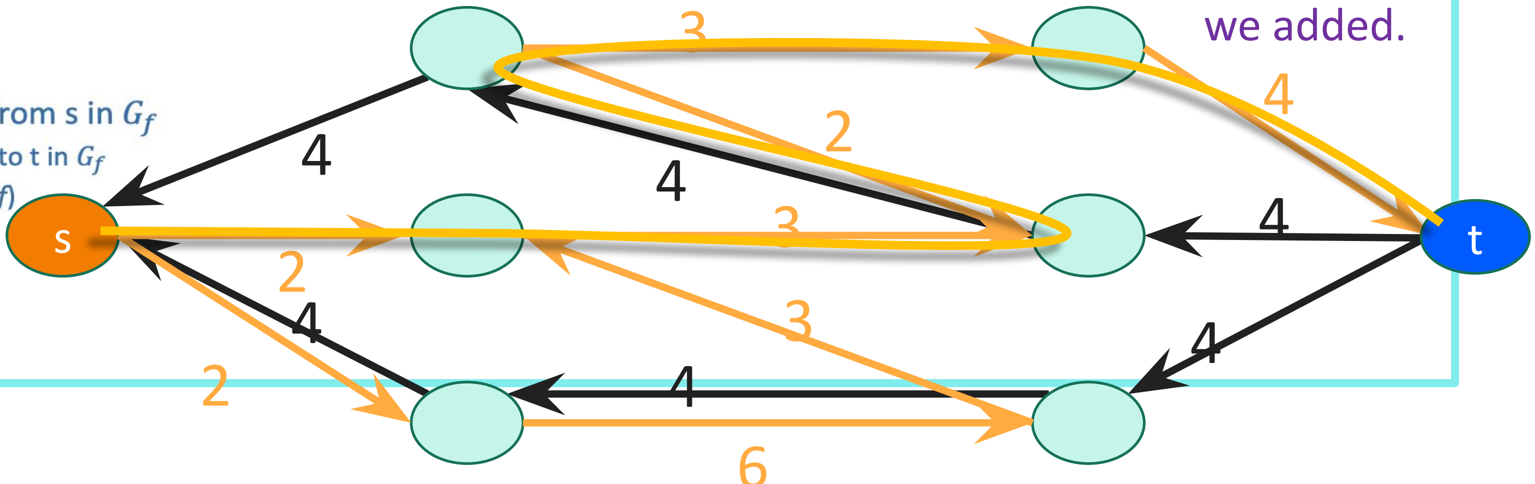
Example of Ford-Fulkerson

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



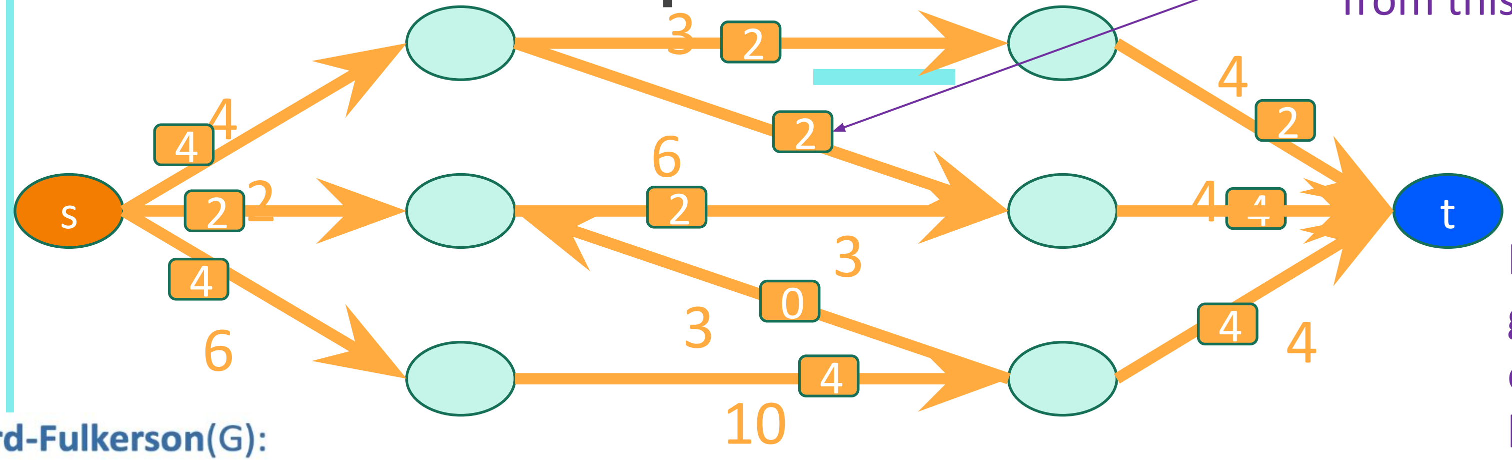
Notice that we're going back along one of the backwards edges we added.

- **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
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f

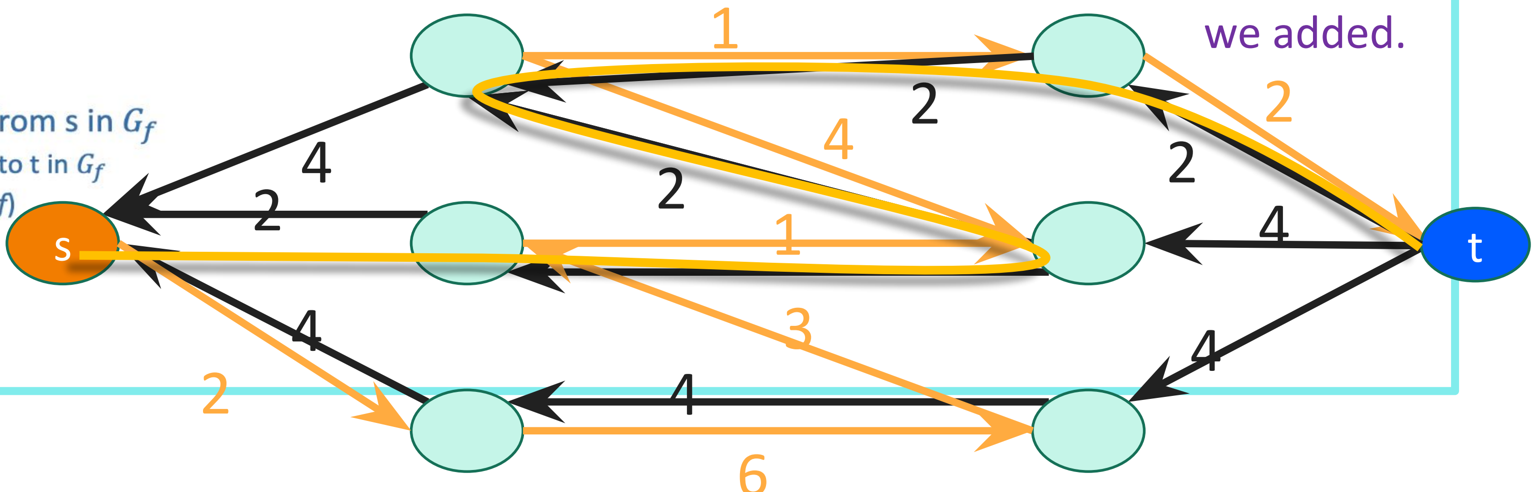


Example of Ford-Fulkerson

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



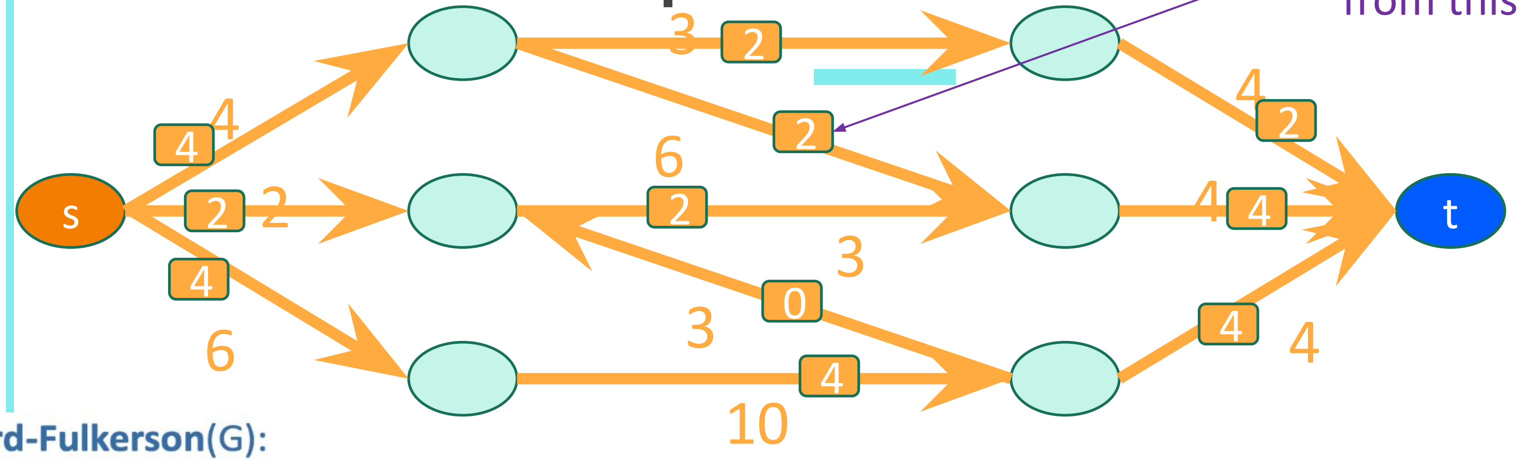
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



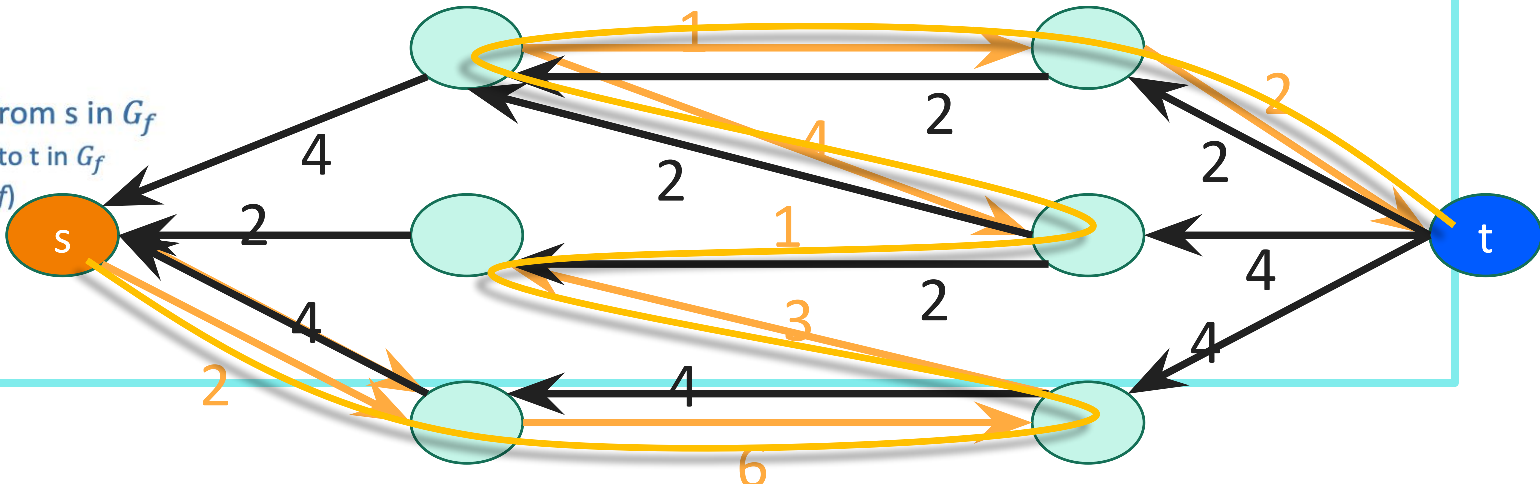
Notice that we're going back along one of the backwards edges we added.

Example of Ford-Fulkerson

We will remove flow from this edge AGAIN.

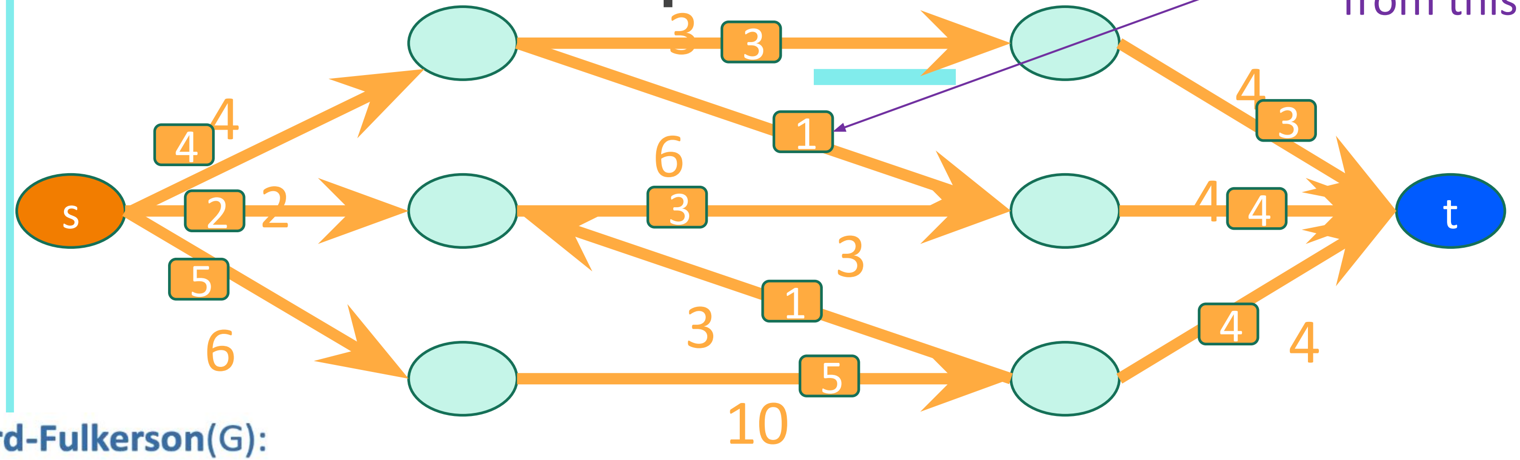


- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f

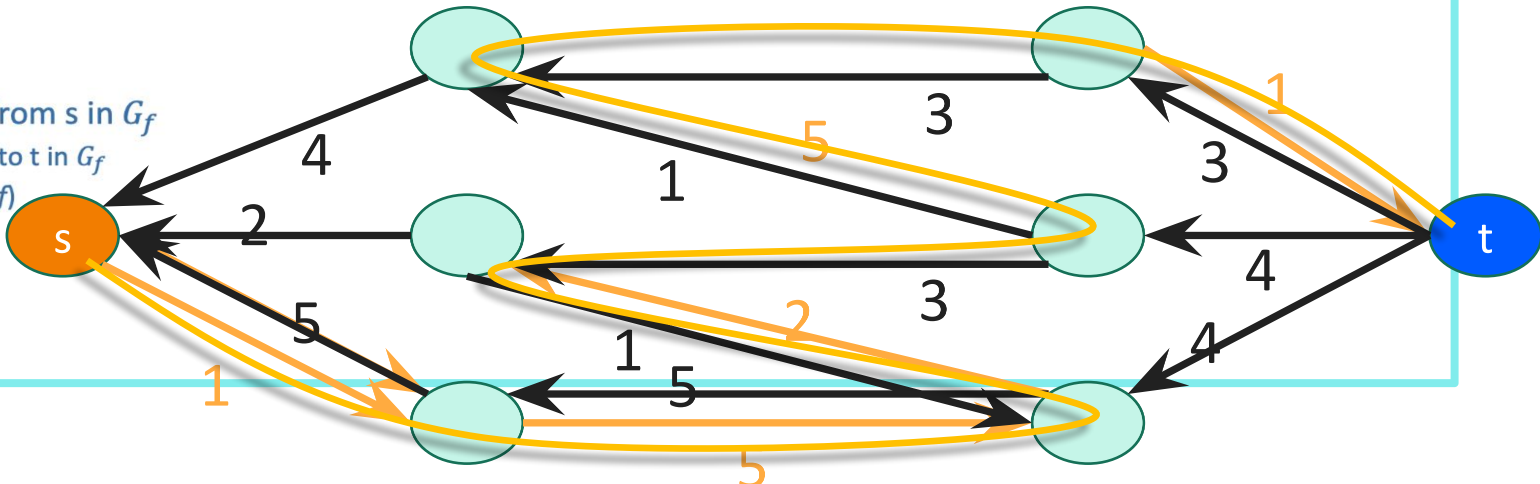


Example of Ford-Fulkerson

We will remove flow from this edge AGAIN.

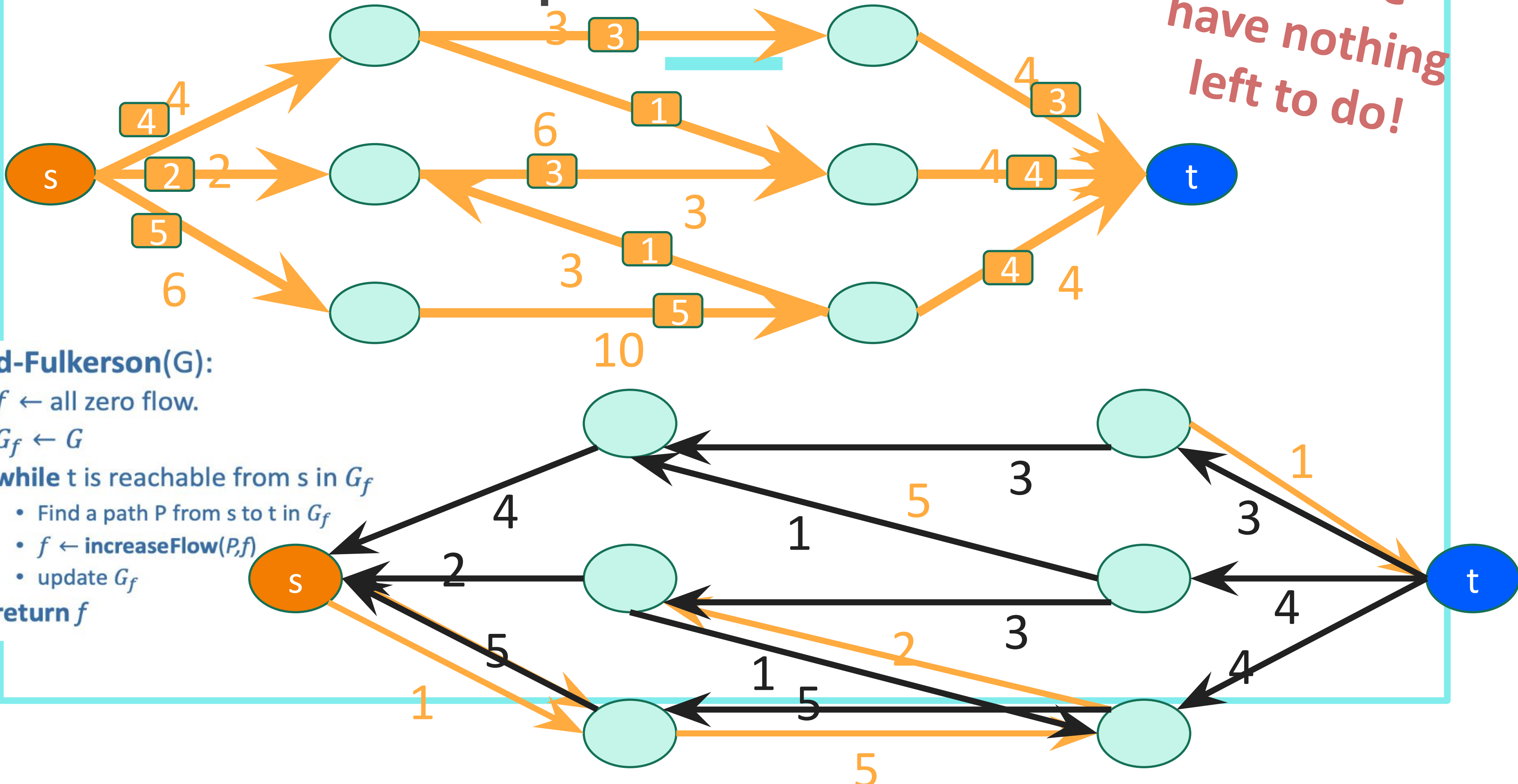


- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f



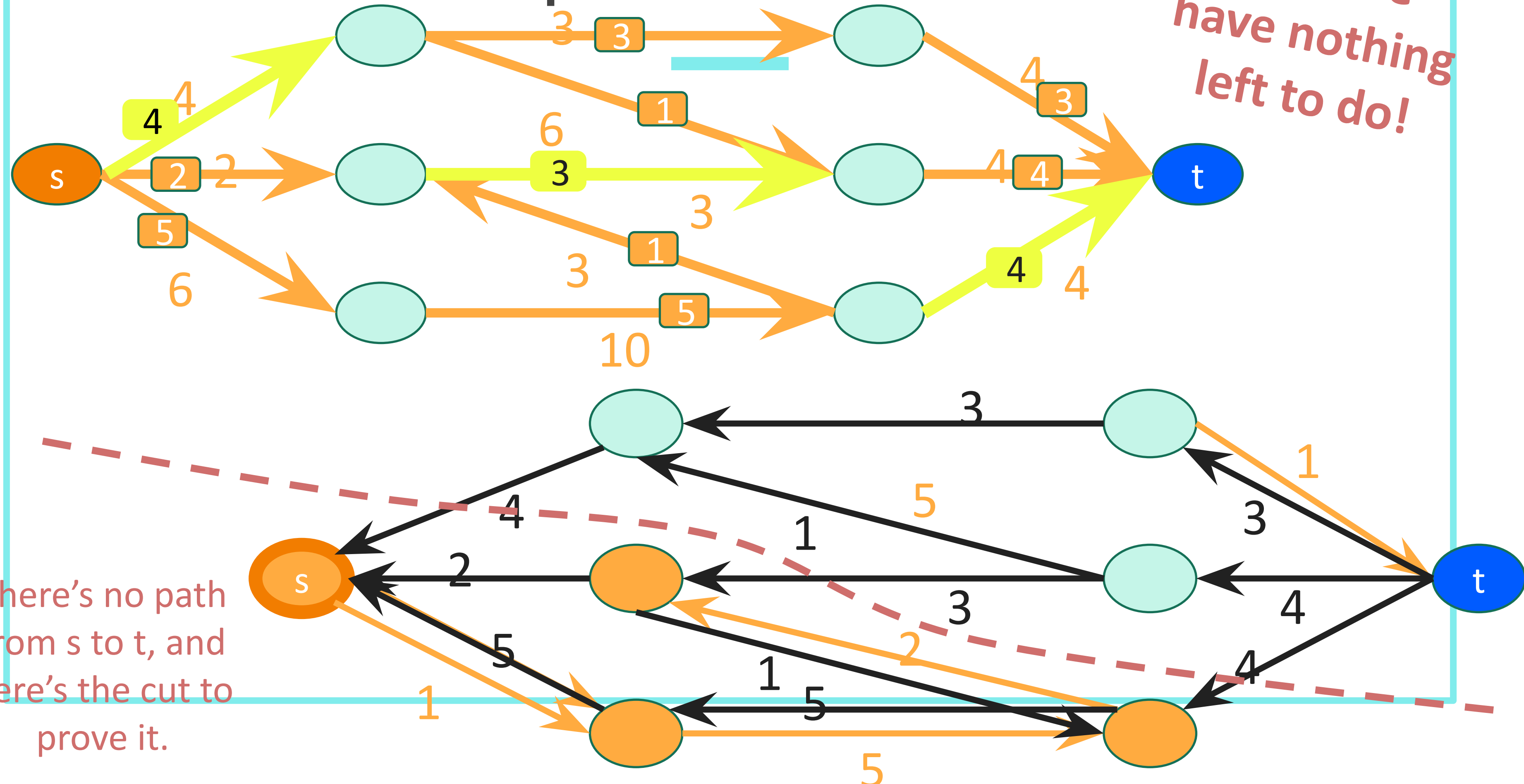
Example of Ford-Fulkerson

Now we have nothing left to do!



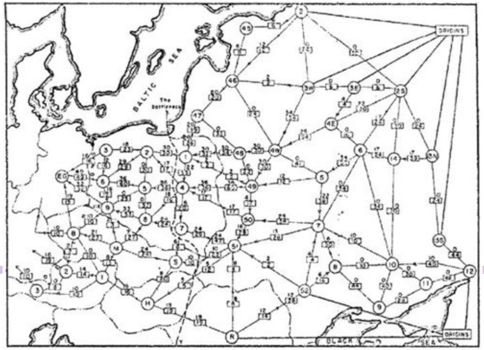
- **Ford-Fulkerson**(G):
  - $f \leftarrow$ all zero flow.
  - $G_f \leftarrow G$
  - **while** t is reachable from s in G_f
    - Find a path P from s to t in G_f
    - $f \leftarrow$ **increaseFlow**(P,f)
    - update G_f
  - **return** f

Example of Ford-Fulkerson



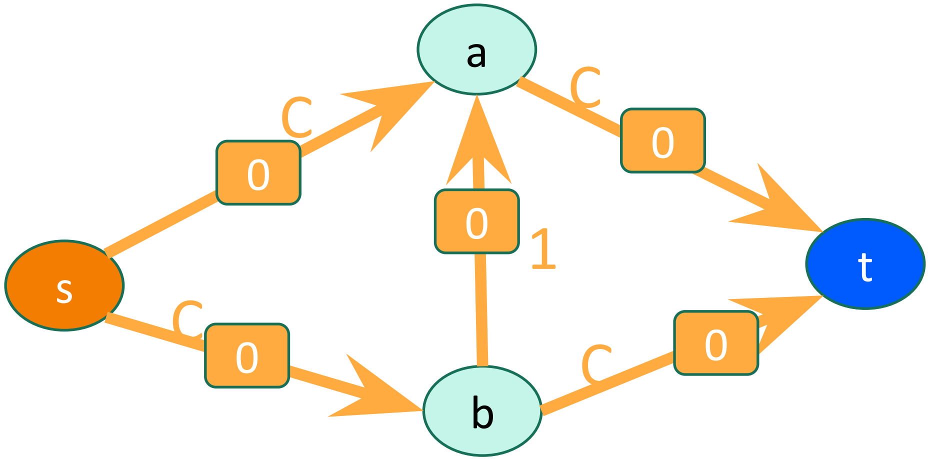
What have we learned?

- 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???

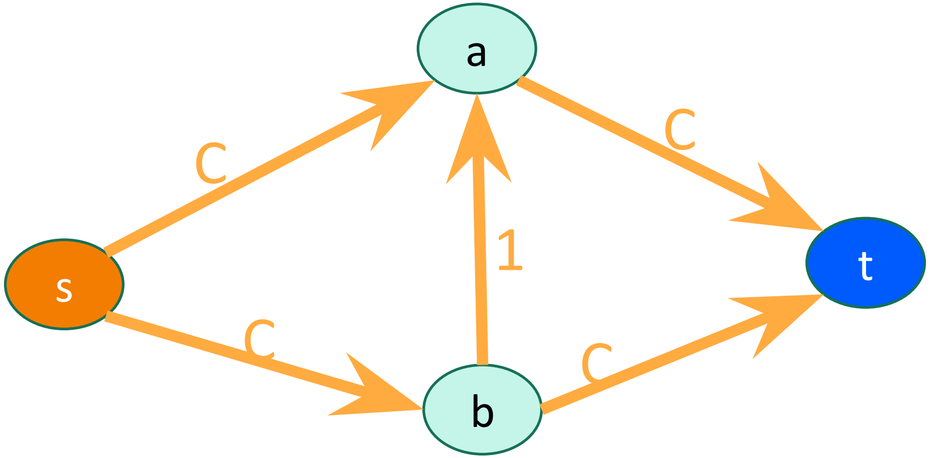


Why should we be concerned?

Suppose we just picked paths arbitrarily.



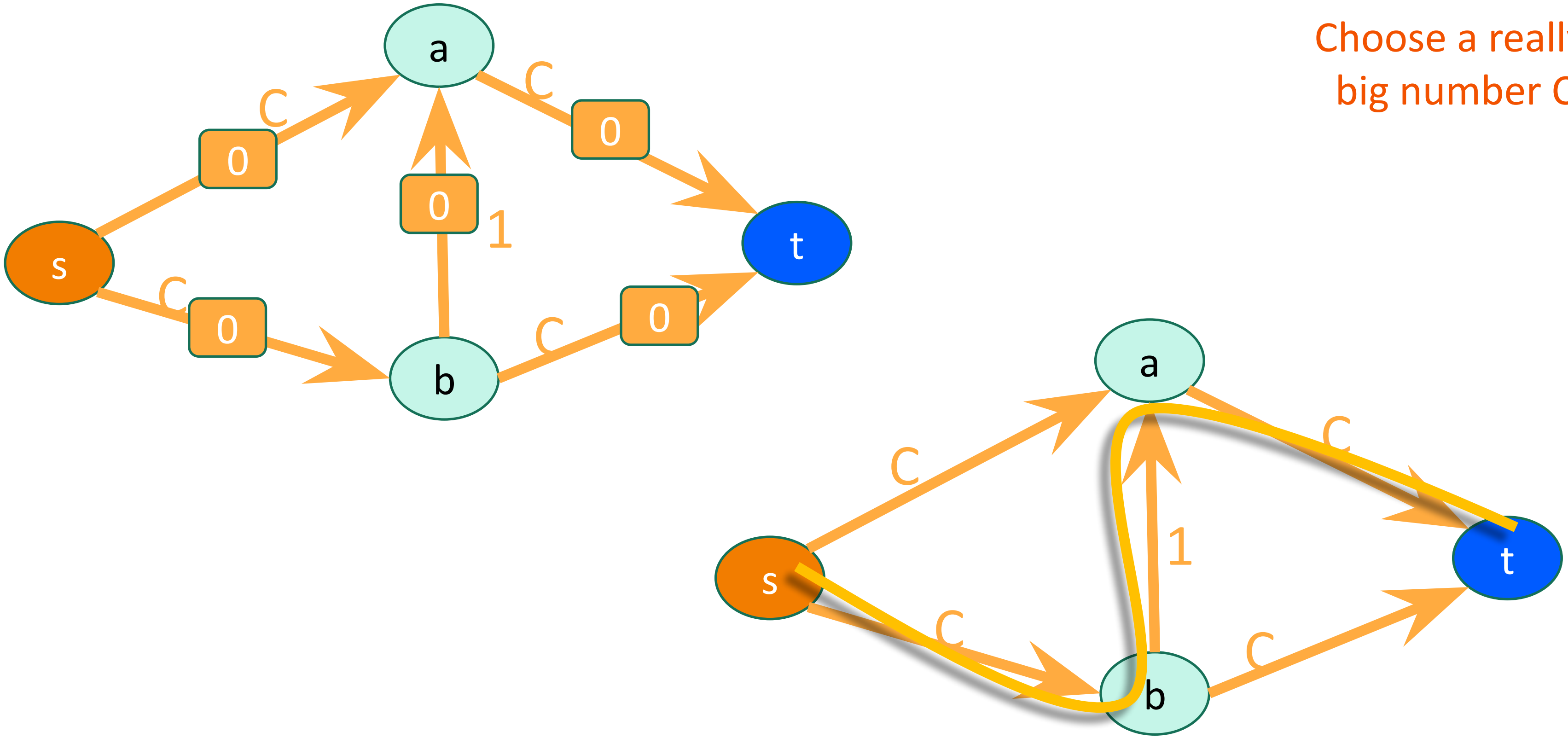
Choose a really big number C.



Why should we be concerned?

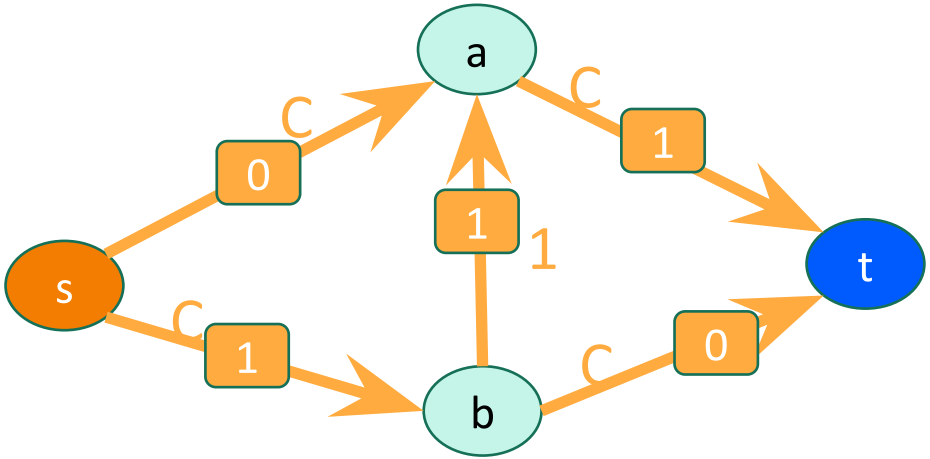
Suppose we just picked paths arbitrarily.

Choose a really big number C.

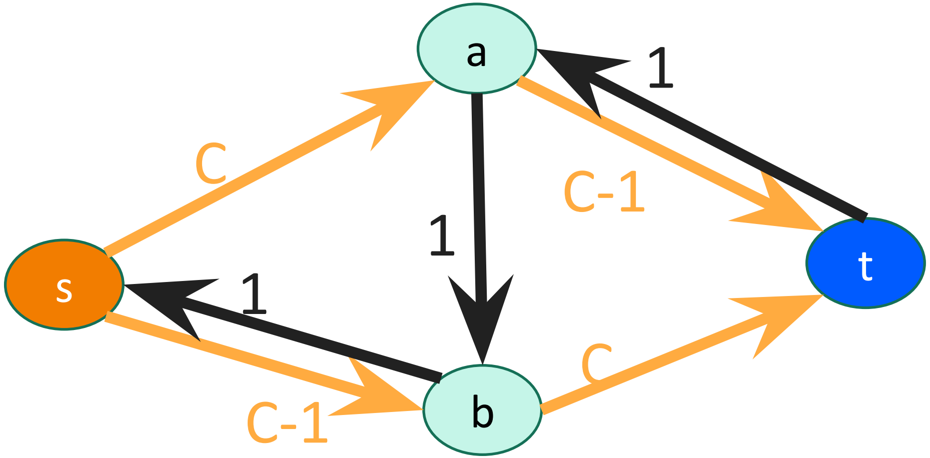


Why should we be concerned?

Suppose we just picked paths arbitrarily.



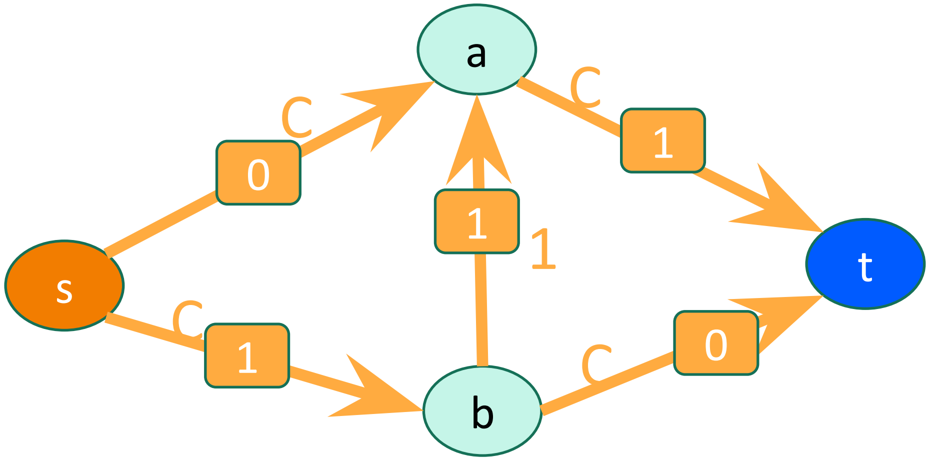
Choose a really big number C.



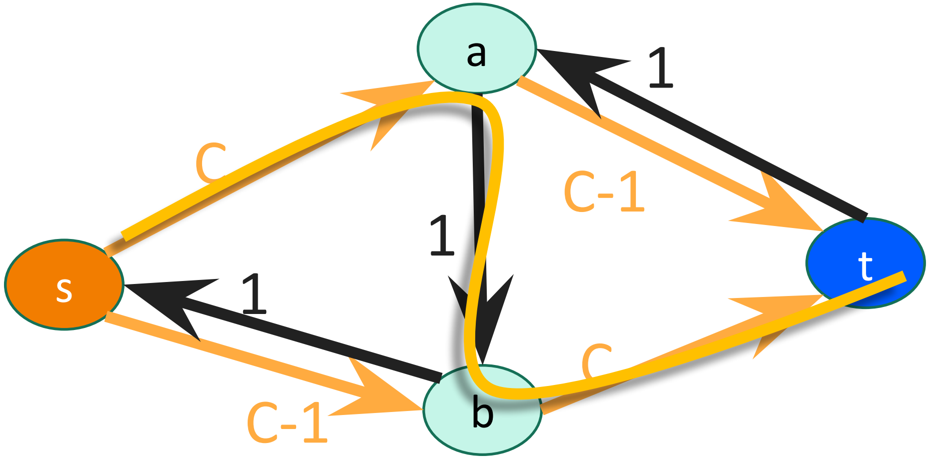
The edge (b,a) disappeared from the residual graph!

Why should we be concerned?

Suppose we just picked paths arbitrarily.

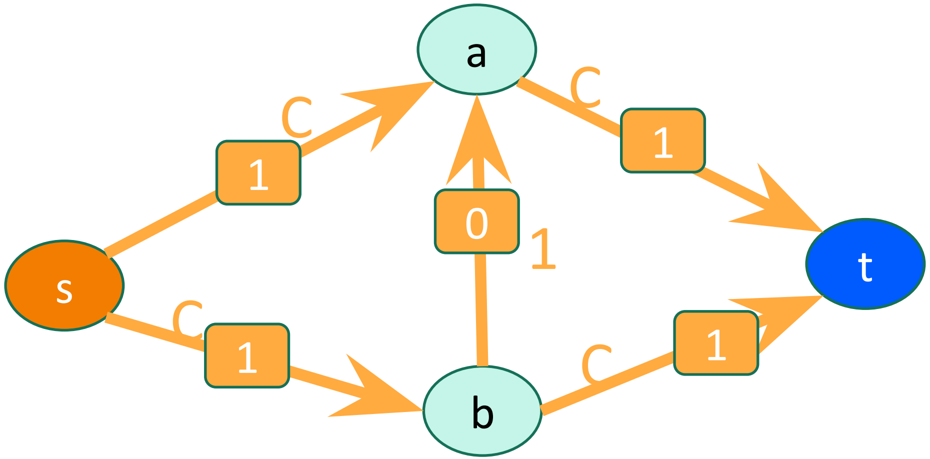


Choose a really big number C.

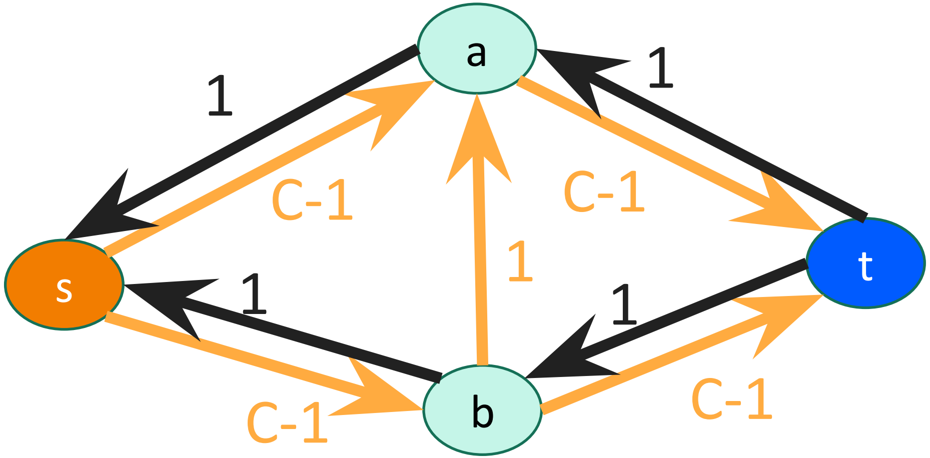


Why should we be concerned?

Suppose we just picked paths arbitrarily.



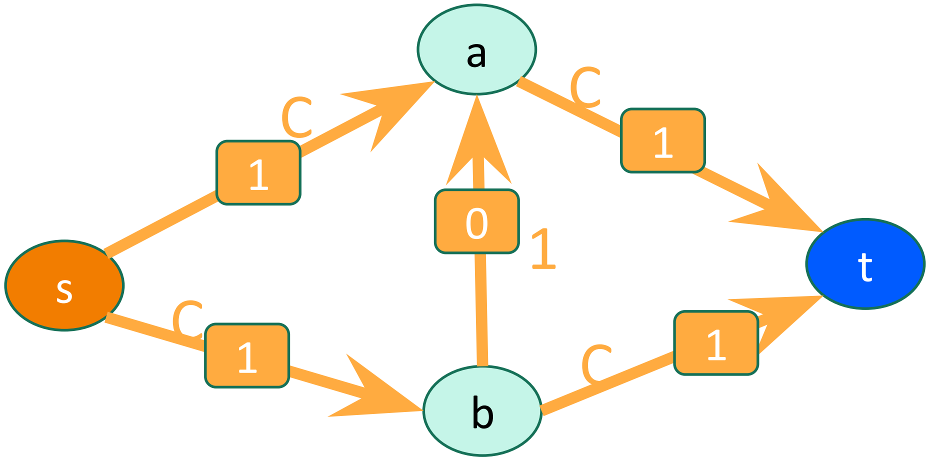
Choose a really big number C.



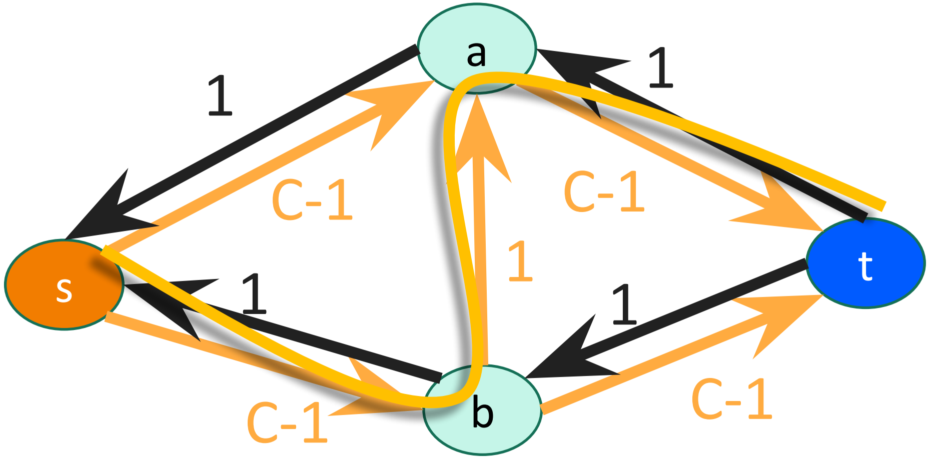
The edge (b,a) re-appeared in the residual graph!

Why should we be concerned?

Suppose we just picked paths arbitrarily.

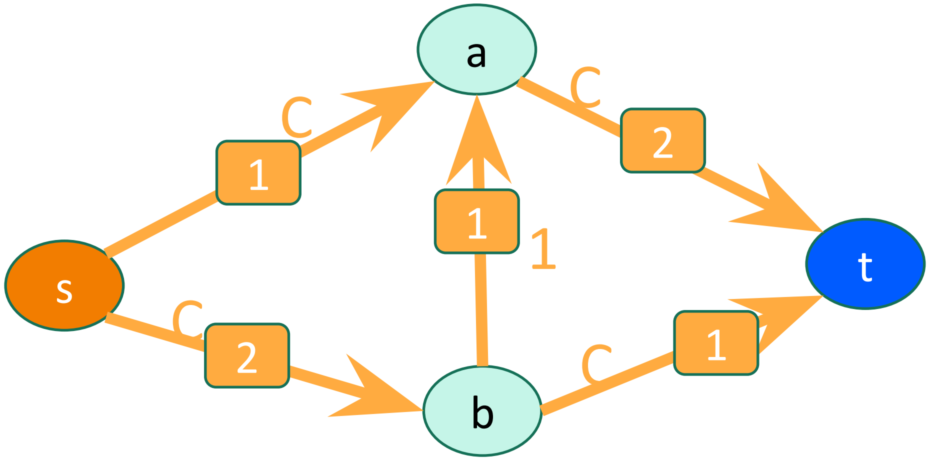


Choose a really big number C.

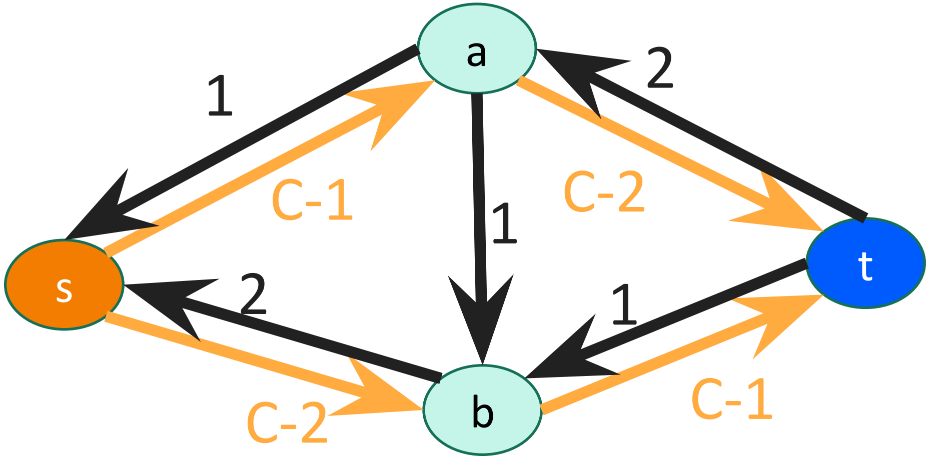


Why should we be concerned?

Suppose we just picked paths arbitrarily.



Choose a really big number C.

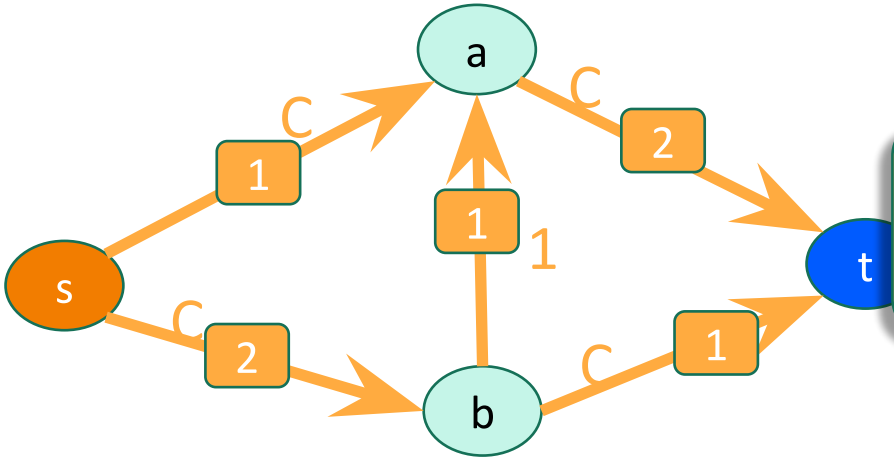


The edge (b,a) disappeared from the residual graph!

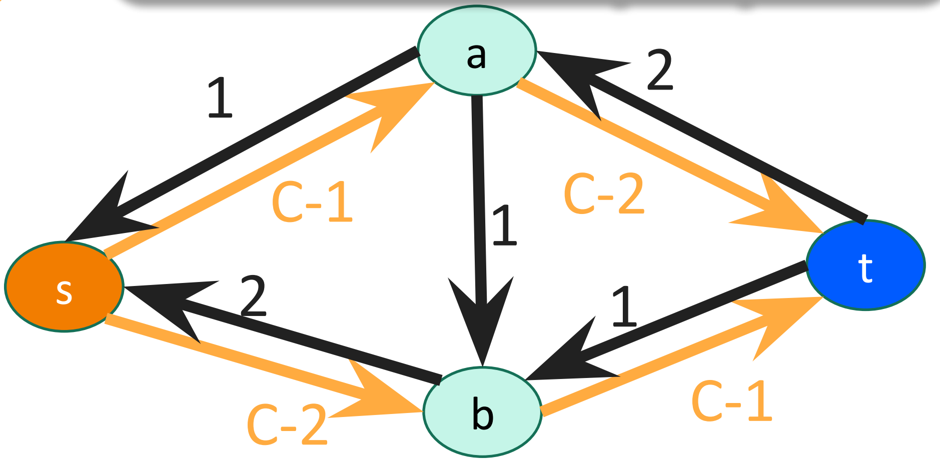
Why should we be concerned?

Suppose we just picked paths arbitrarily.

Choose a really big number C.



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



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

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 Ford-Fulkerson with BFS leads to the **Edmonds-Karp algorithm.**
    - It turns out this will work in time $O(nm^2)$.
    - (That's not the only way to do it!)

Running time

- Edmonds-Karp algorithm (aka Ford-Fulkerson with BFS) runs in time $O(nm^2)$.
- 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 observation…

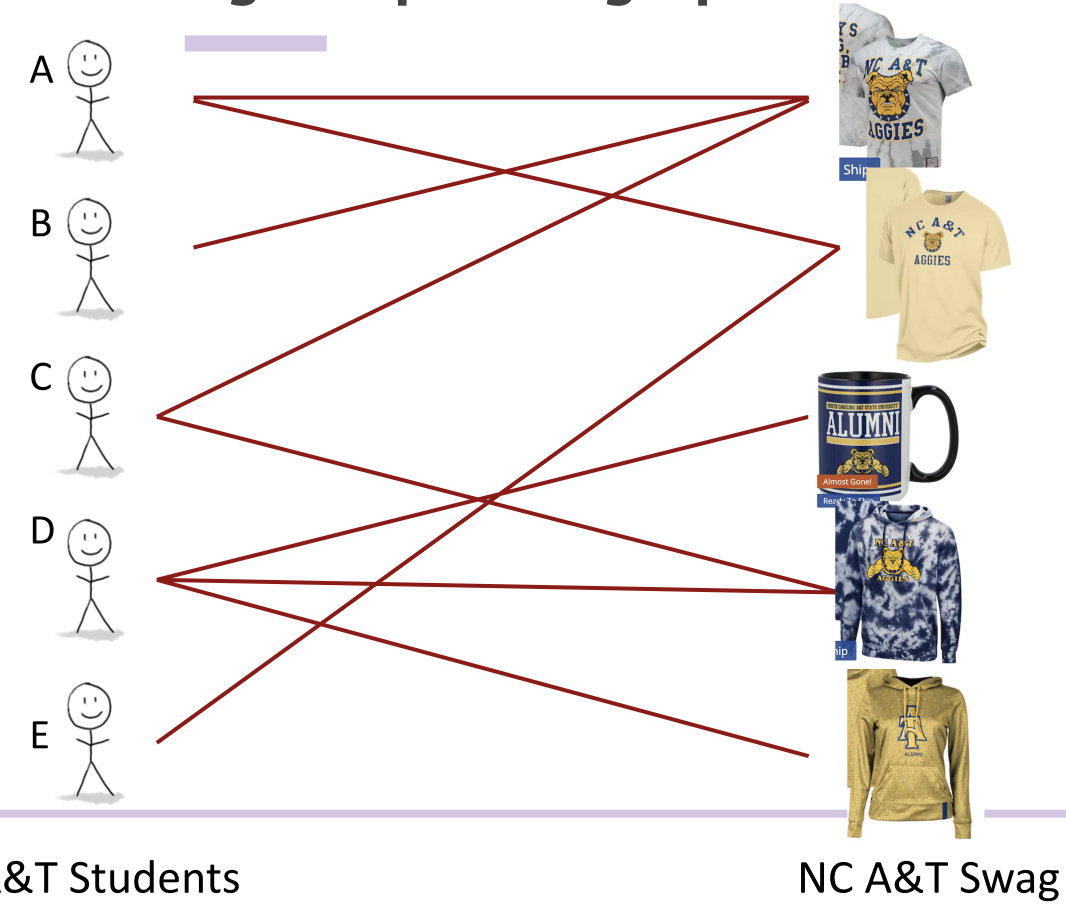
If all the capacities are integers, then the flows in any max flow are also all integers.

- When we update flows in Ford-Fulkerson, we're only ever adding or subtracting integers.
- Since we started with 0 (an integer), everything stays an integer.

We'll see why in a second…

Maximum matching in bipartite graphs

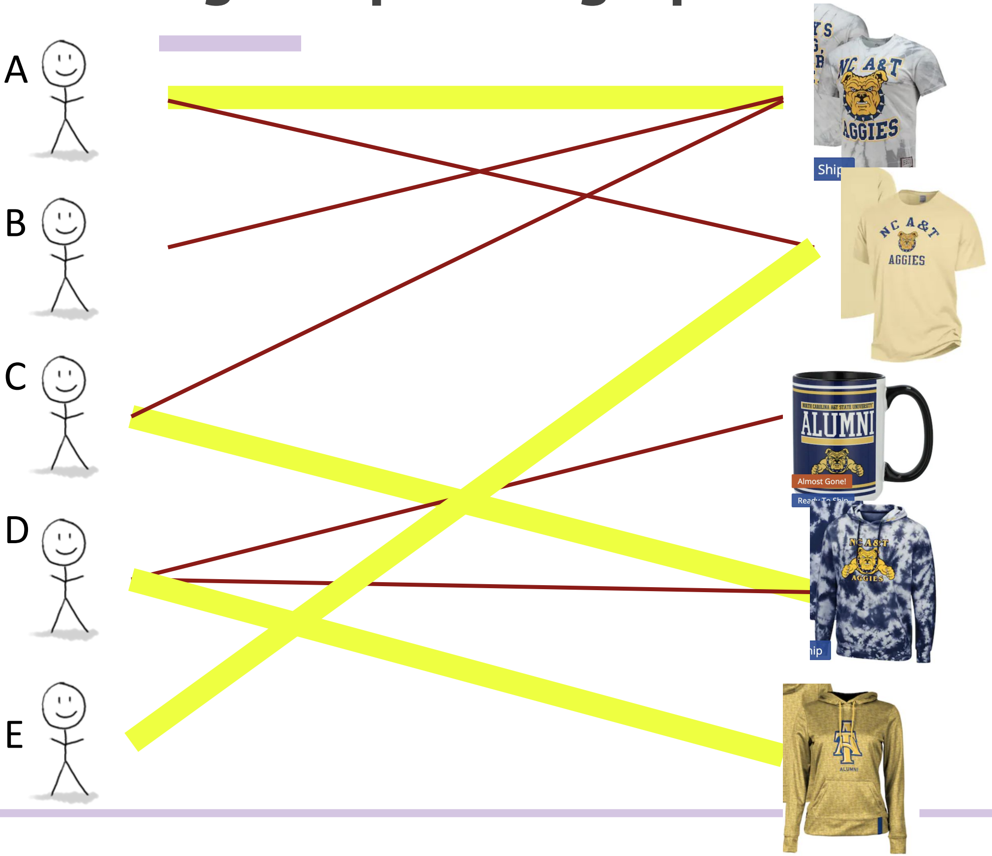
- Different students only want certain items of NCAT swag (depending on fit, style, etc.)
- **How can we make as many students as possible happy?**



Maximum matching in bipartite graphs

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

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

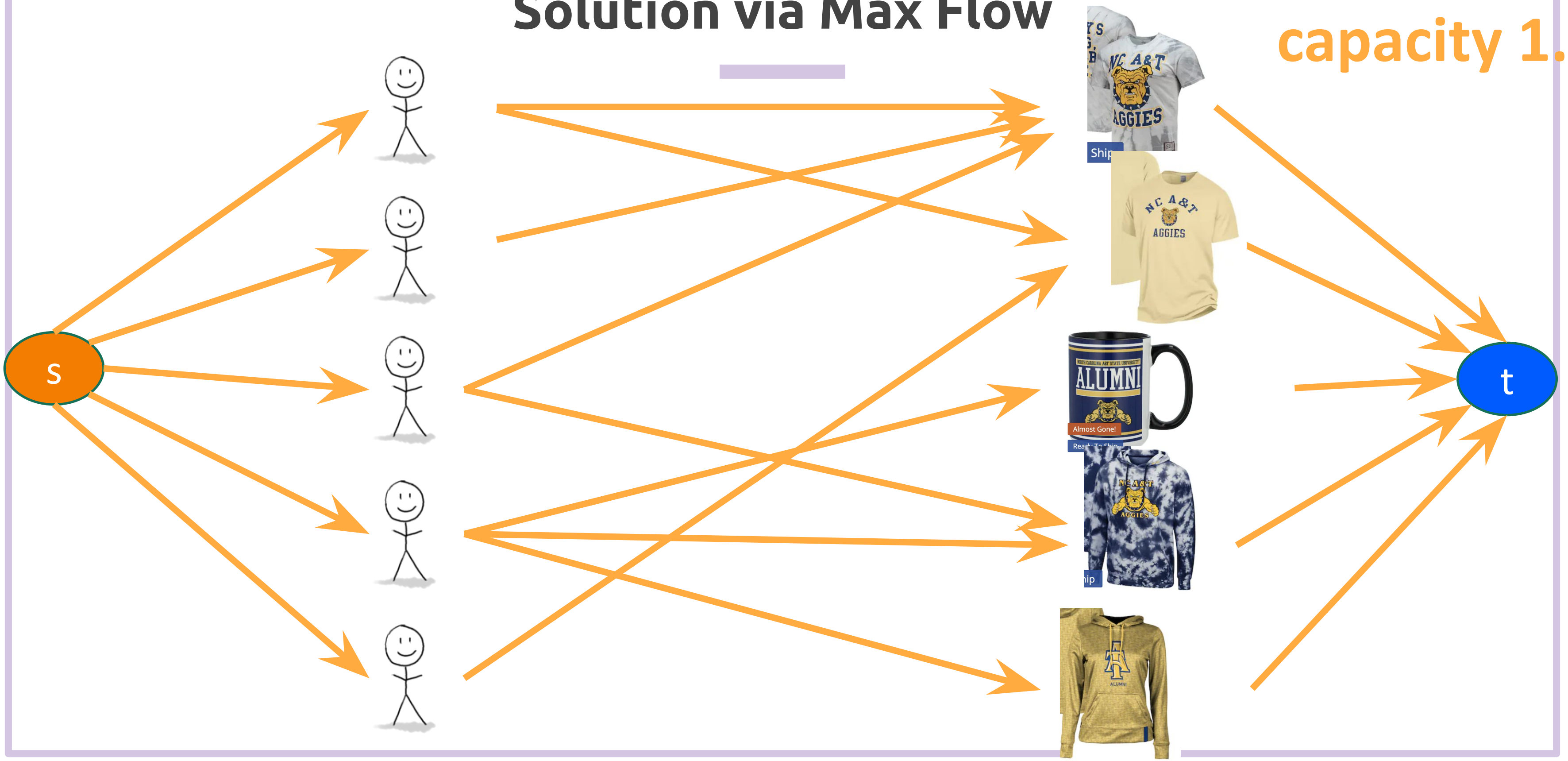


NC A&T Students

NC A&T Swag

Solution via Max Flow

All edges have capacity 1.

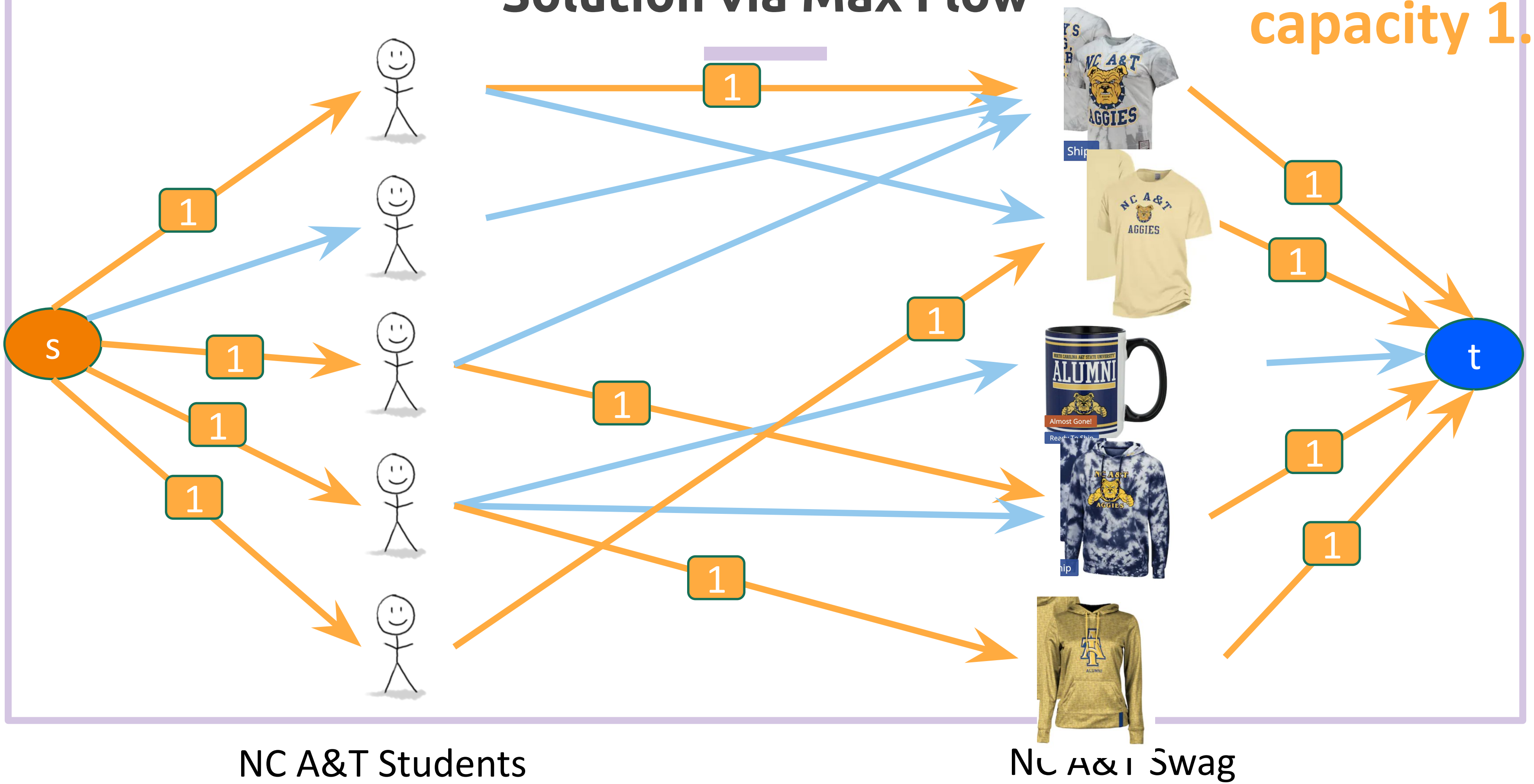


NC A&T Students

NC A&T Swag

Solution via Max Flow

All edges have capacity 1.

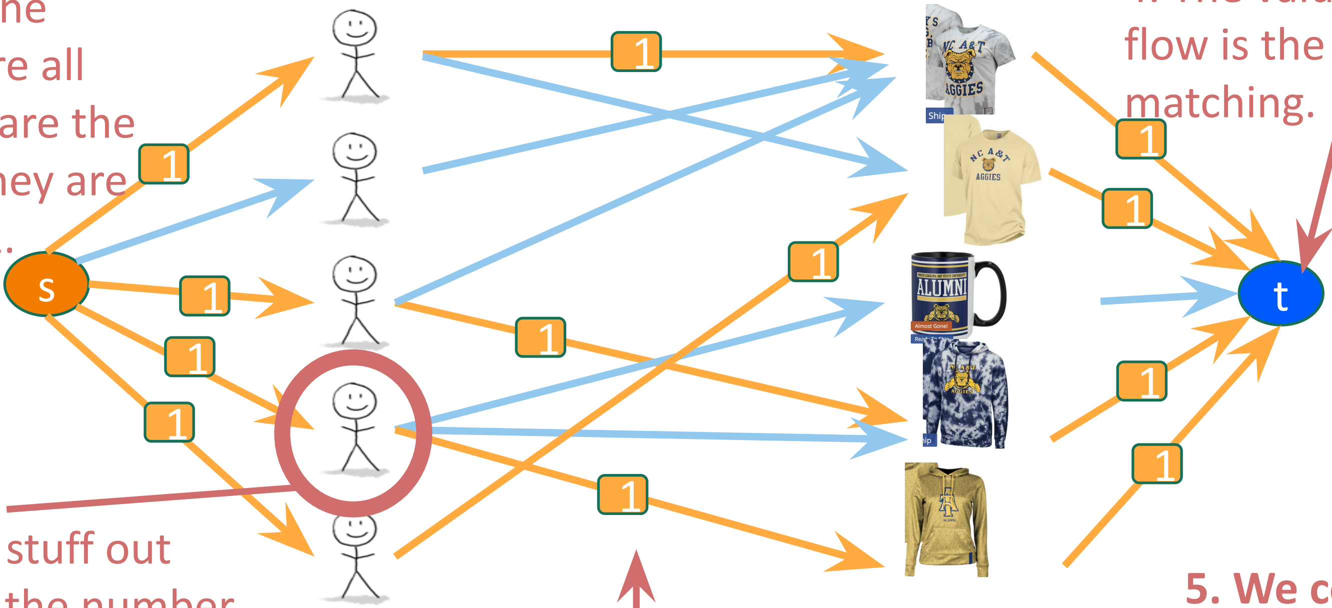


Solution via Max Flow: Why does it work?

All edges have capacity 1.

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

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



Value of this flow is 4.

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

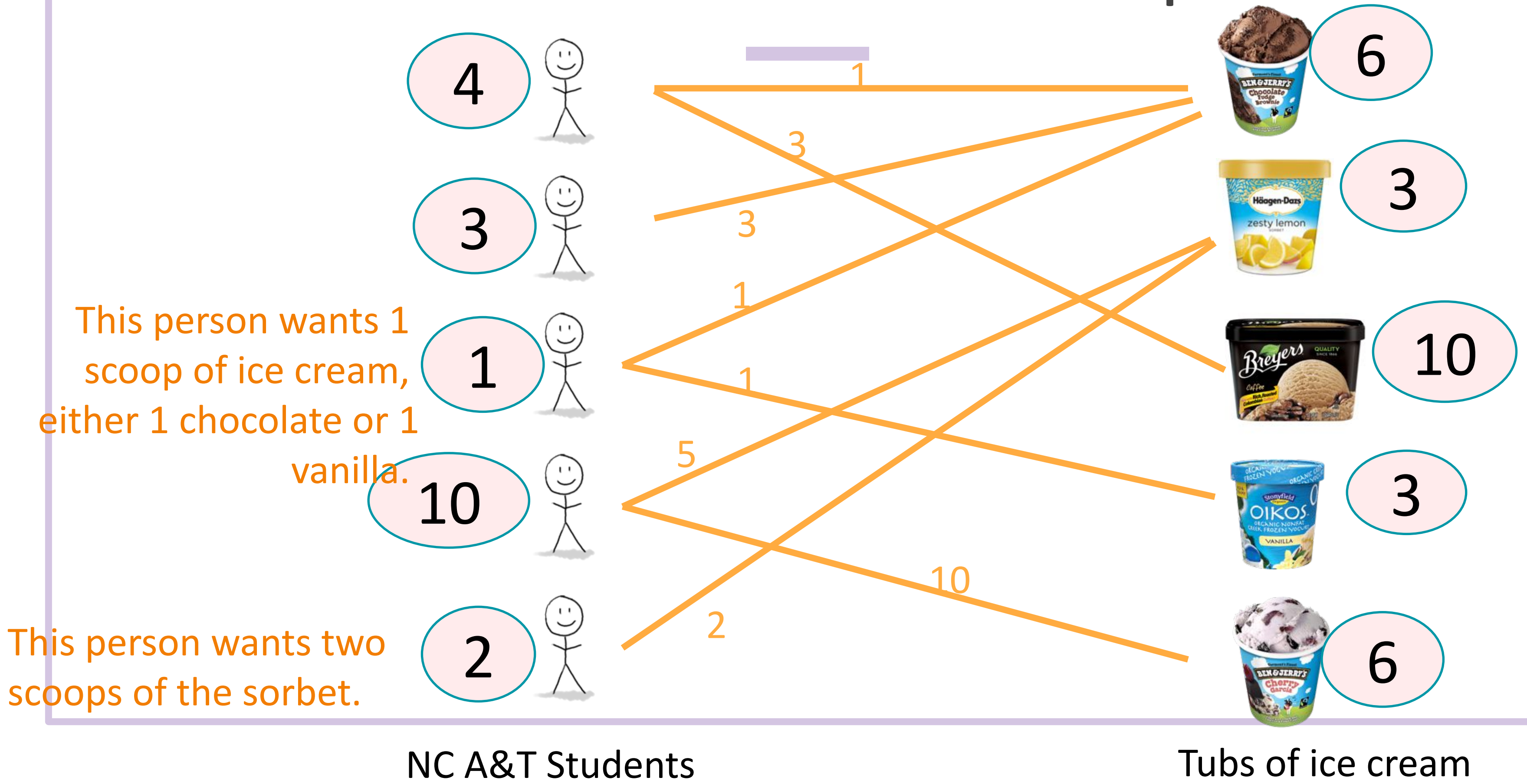
3. Thus, the edges with flow on them form a matching. (And, any matching gives a flow).

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

More complicated example: optimal assignment problems

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

How can we serve as much ice cream as possible?



Solution via max flow

NC A&T Students

Tubs of ice cream

Solution via max flow

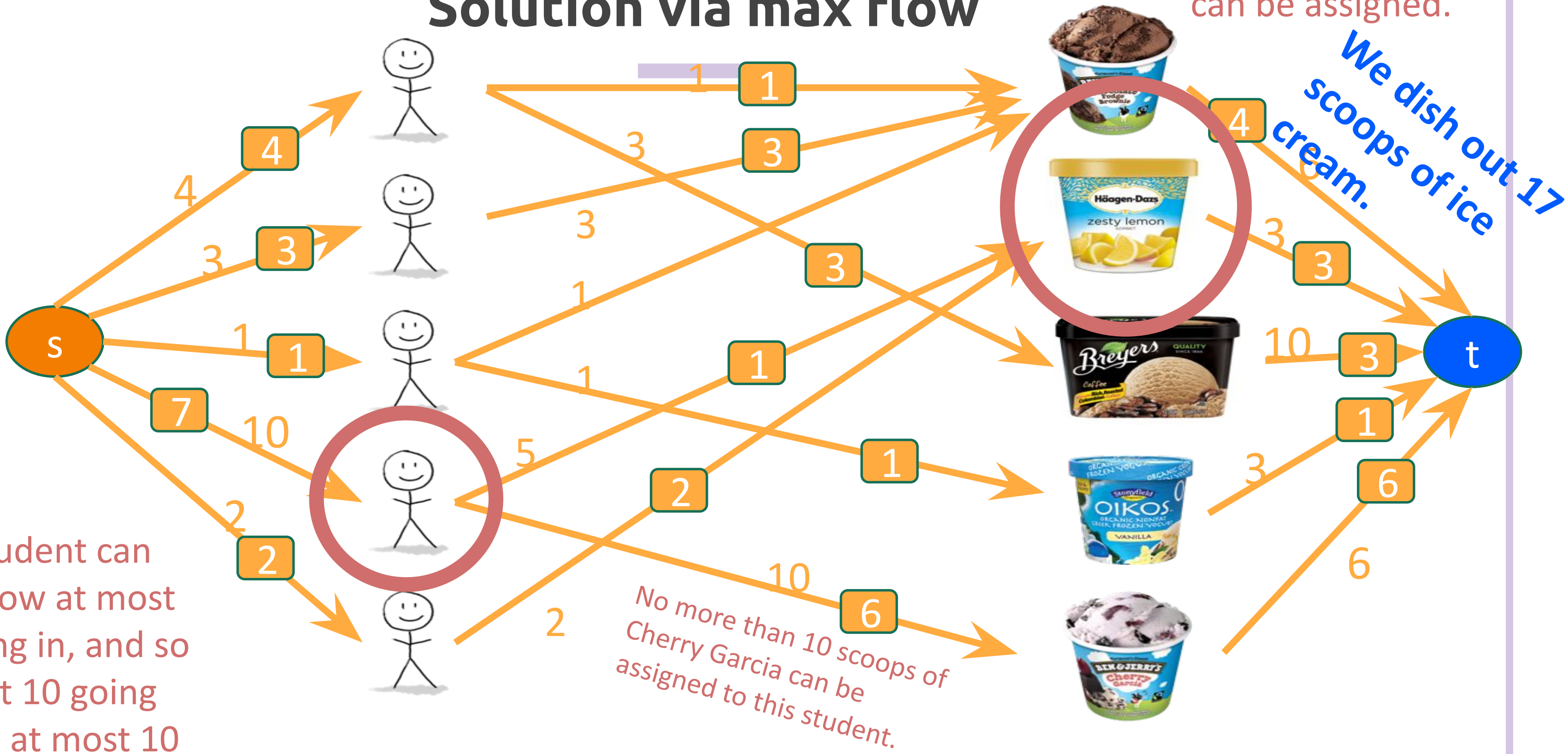
Give this person 1 scoop of this ice cream.

NC A&T Students

Tubs of ice cream

Solution via max flow

No more than 3 scoops of sorbet can be assigned.



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

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

What have we learned?

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

Recap

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

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 18: Netflow Flow

Lecturer: Chris Lucas (cflucas@ncat.edu)