طراحی الگوریتم ها (CE221)

جلسه هجدهم:
شاره بیشینه

سجاد شیرعلی شهرضا

بهار 1401

شنبه، 31 اردیبهشت 1401

Based on Stanford CS161 Slides from Summer 2021 by Karey Shi

اطلاع رسانی

- بخش مرتبط کتاب برای این جلسه: 26
- یادآوری مهلت ارسال تمرین سوم: 8 صبح روز دوشنبه 9 خرداد 1401

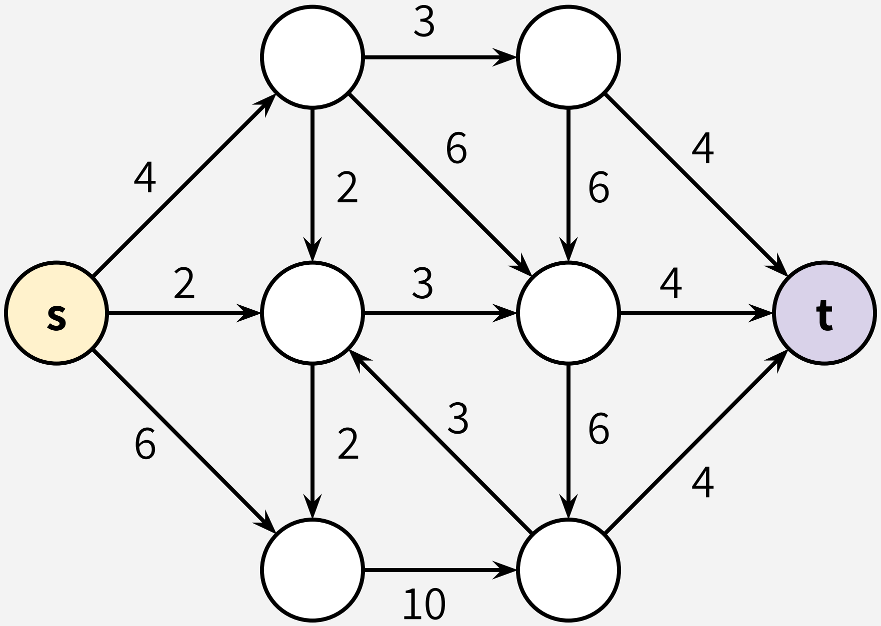
برش کمینه s-t

برش کمینه برای جداسازی دو راس خاص

s–t MINIMUM CUTS

A **minimum s–t cut** is a cut which separates **s** from **t** with minimum cost

Now, we're talking about ***directed & weighted*** graphs.



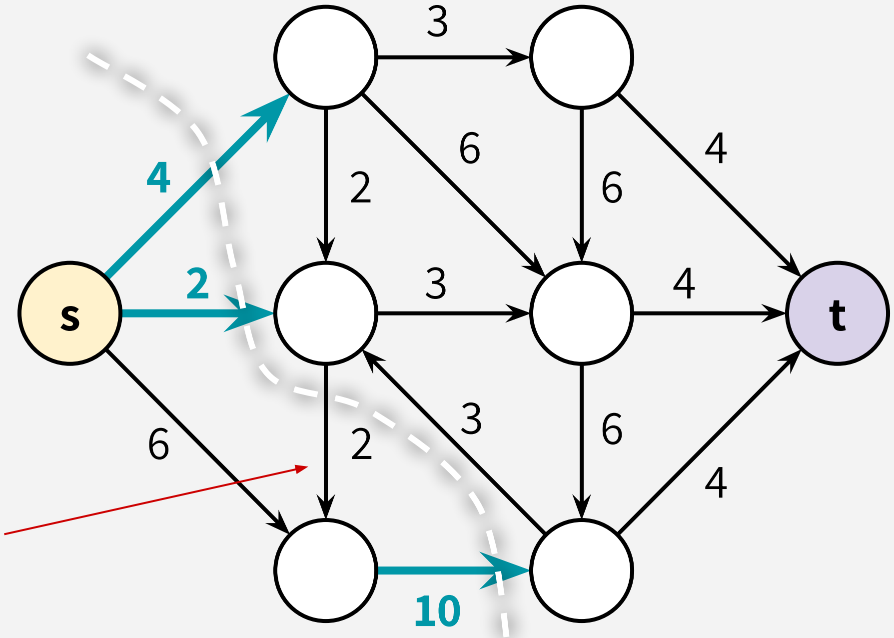
The **cost**/**capacity** of a cut is the sum of the capacities of the edges that *cross* the cut

(i.e. edges that go **from** the s-side to the t-side)

s–t MINIMUM CUTS

A **minimum s–t cut** is a cut which separates **s** from **t** with minimum cost

This is a cut that separates **s** from **t**! It has cost 4 + 2 + 10 = 16



Note that this edge does *not* cross the cut! It's going in the wrong direction (from the t-side to the s-side)

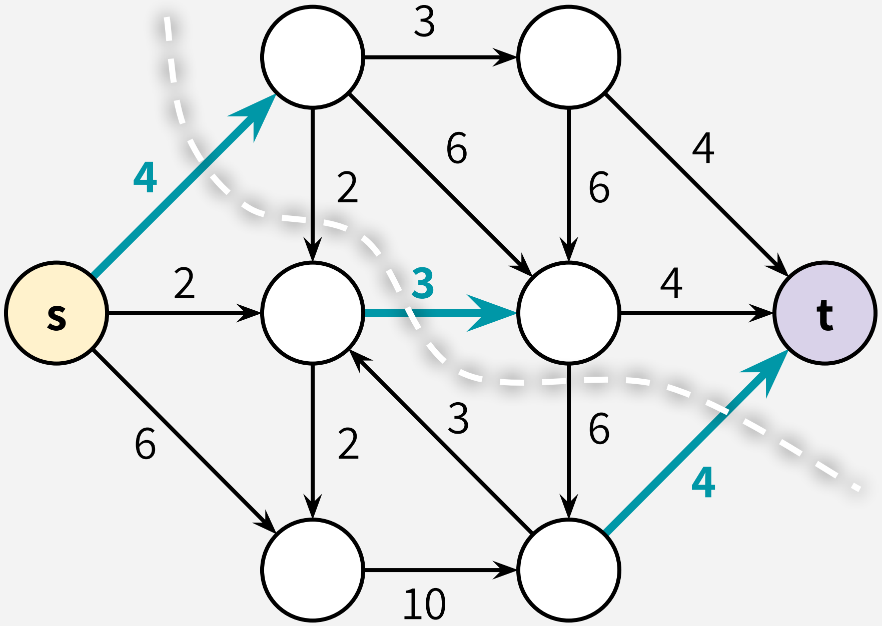
The **cost**/**capacity** of a cut is the sum of the capacities of the edges that *cross* the cut (i.e. edges that go **from** the s-side to the t-side)

s–t MINIMUM CUTS

A **minimum s–t cut** is a cut which separates **s** from **t** with minimum cost

This is a cut that separates **s** from **t**!
It has cost
4 + 3 + 4 = **11**

This is actually a minimum s–t cut!



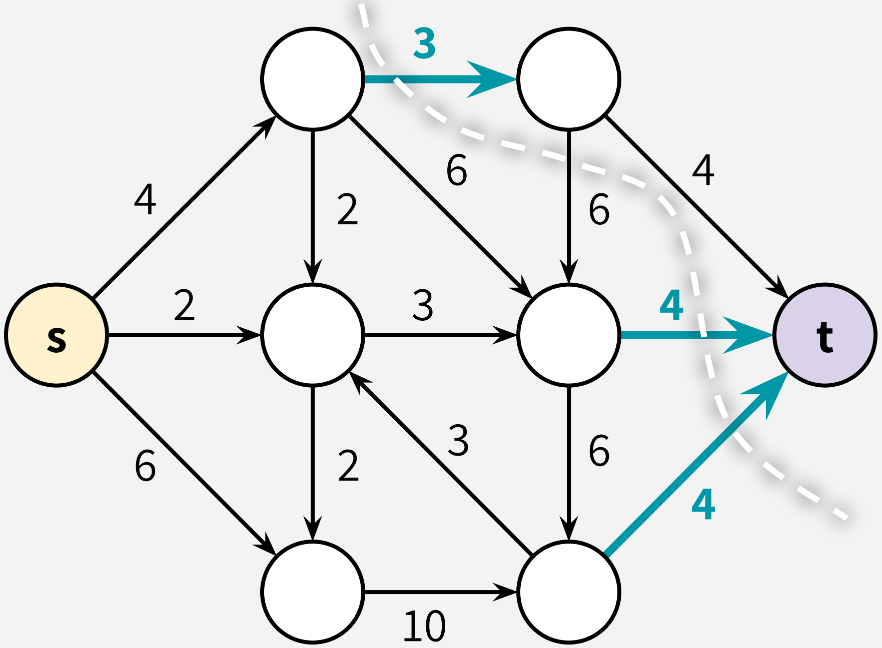
The **cost**/**capacity** of a cut is the sum of the capacities of the edges that *cross* the cut
(i.e. edges that go **from** the s-side to the t-side)

s–t MINIMUM CUTS

A **minimum s–t cut** is a cut which separates **s** from **t** with minimum cost

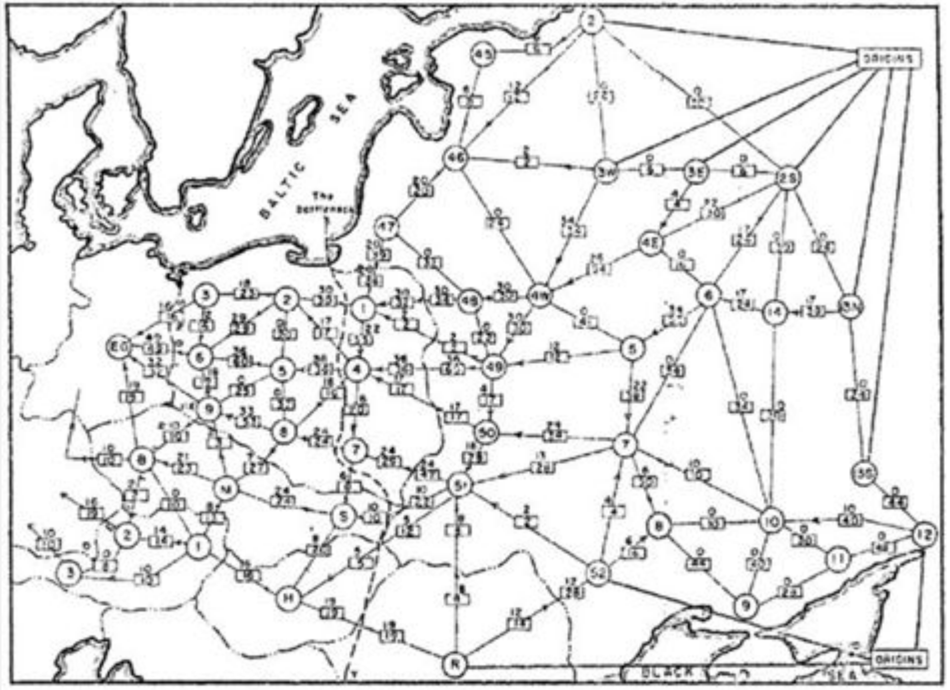
This cut has cost
3 + 4 + 4 = **11**

This is also a minimum s–t cut!



The **cost**/**capacity** of a cut is the sum of the capacities of the edges that *cross* the cut
(i.e. edges that go **from** the s-side to the t-side)

EXAMPLE APPLICATION



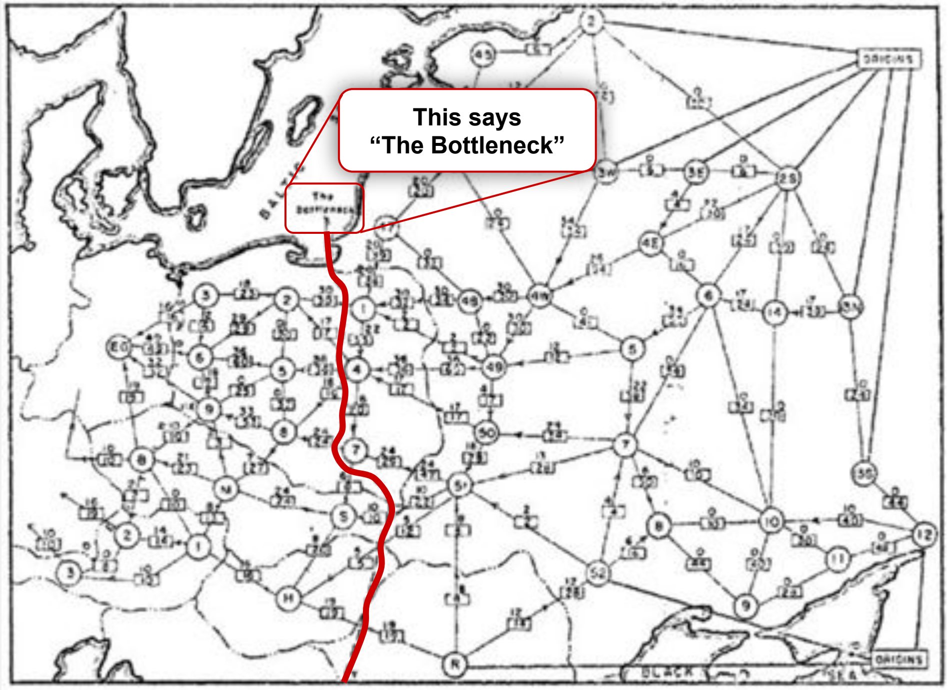
1955 map of rail networks from the Soviet Union to Eastern Europe.

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.

EXAMPLE APPLICATION



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.

EXAMPLE APPLICATION

1955 map of rail networks from the Soviet Union to Eastern Europe.

Declassified in 1999.

44 edges, 105 vertices

How do we find a minimum s–t cut?

First, we need to talk about flows!

f routes from
stern Europe
ssible.

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

سوال؟

شاره بیشینه

و رابطه آن با برش کمینه

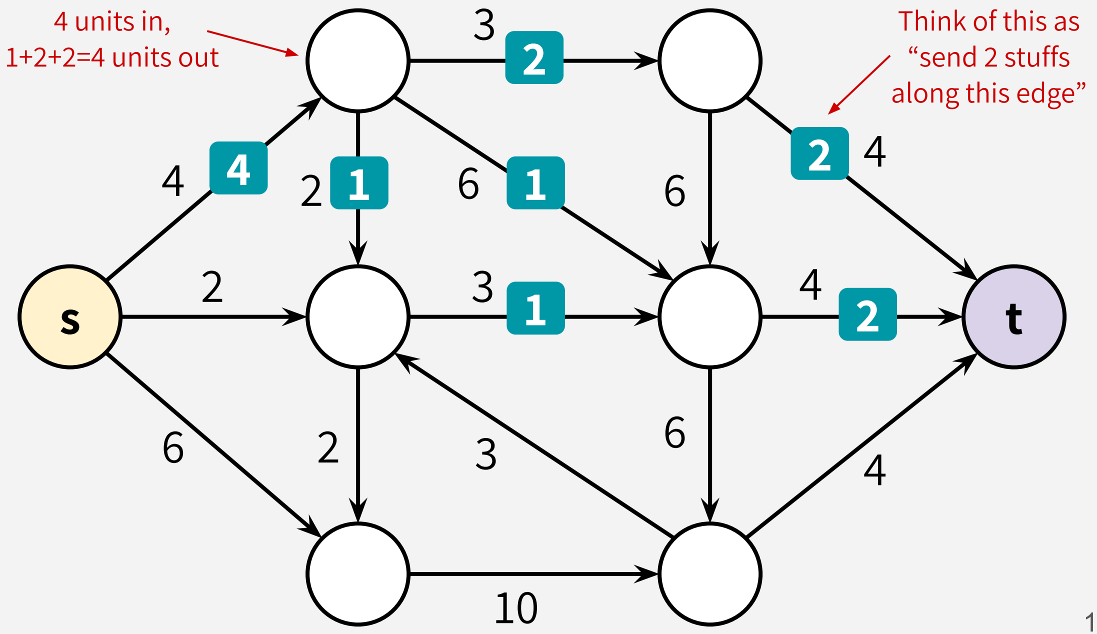
(s–t) MAXIMUM FLOWS

The **value of a flow** is the amount of stuff coming out of **s**
(aka the amount of stuff flowing into **t**, due to flow conservation!)

Every edge has a flow
Edges with 0 flow are unmarked

Capacity Constraint
The flow on any edge must be ≤ its capacity!

Flow Conservation Constraint
At each vertex, the incoming flows must equal the outgoing flows



(s–t) MAXIMUM FLOWS

The **value of a flow** is the amount of stuff coming out of **s**
(aka the amount of stuff flowing into **t**, due to flow conservation!)

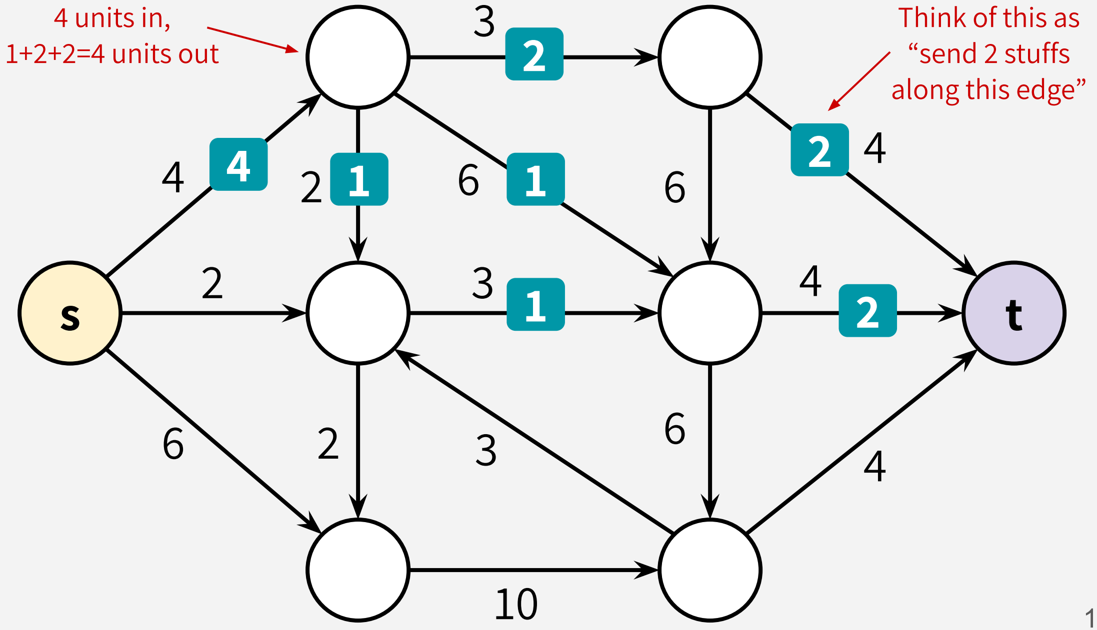
Every edge has a flow
Edges with 0 flow are unmarked

Capacity Constraint

The value of this flow is 4
(Not a max-flow, as it's not utilizing edge capacities well)

Flo int

incoming flows must equal the outgoing flows

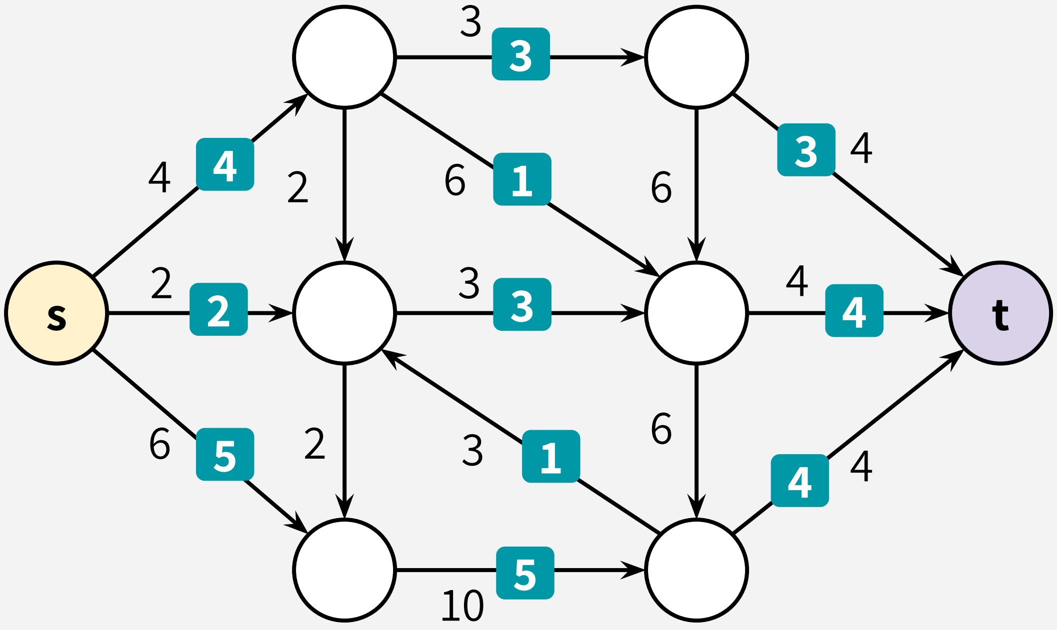


(s–t) MAXIMUM FLOWS

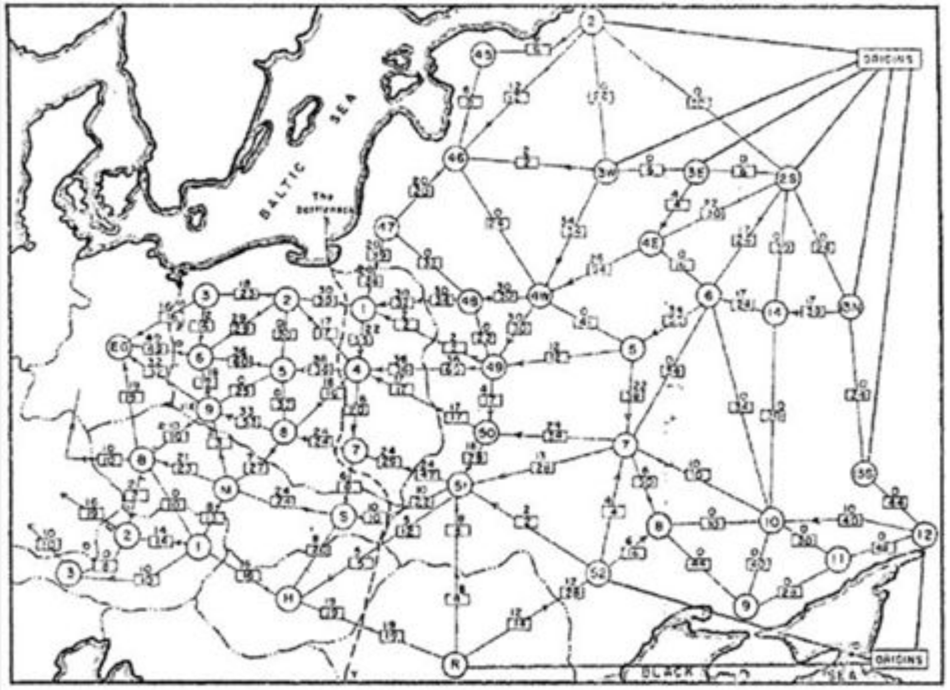
The **value of a flow** is the amount of stuff coming out of **s**
(aka the amount of stuff flowing into **t**, due to flow conservation!)

This one *is* a maximum flow.

The value of this flow is 11.



EXAMPLE APPLICATION

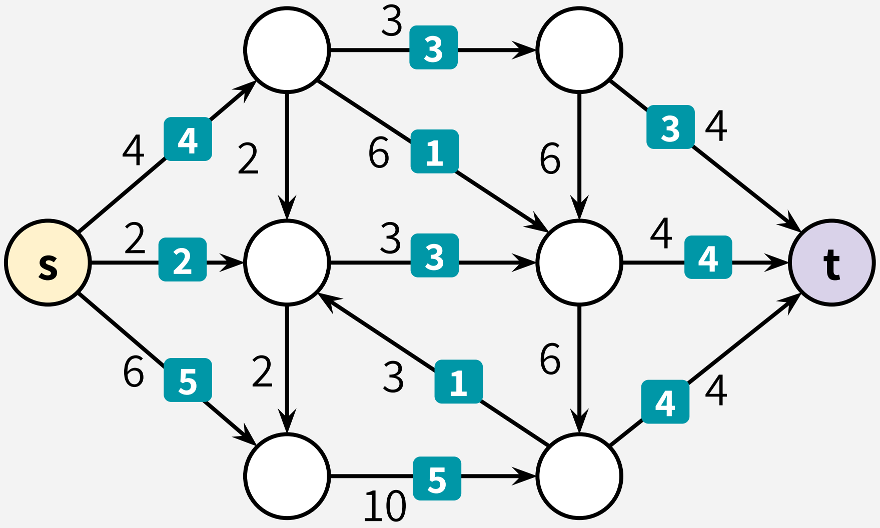


1955 map of rail networks from the Soviet Union to Eastern Europe.
Declassified in 1999.
44 edges, 105 vertices

The Soviet Union wants to route supplies from suppliers in Russia to Eastern Europe as efficiently as possible
(edge capacities/flows are indicated on each edge)

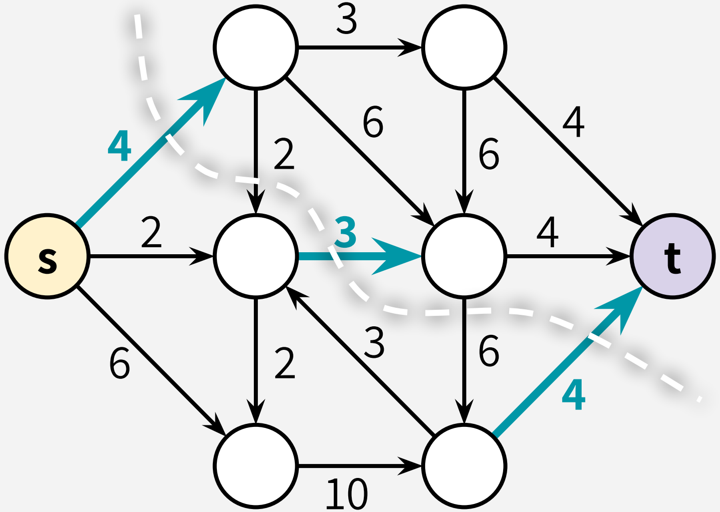
MAX-FLOW MIN-CUT THEOREM

This is not a coincidence!



This max-flow has value 11.

The cost of this min-cut is 11.

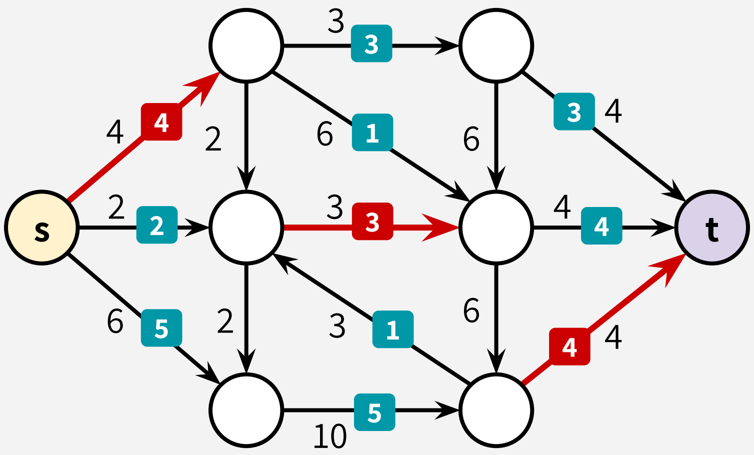


MAX-FLOW MIN-CUT THEOREM

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, edges crossing the min-cut will “fill up”, and this is the bottleneck (once it’s filled up, there’s no way to send more flow from s to t!



سوال؟

MAX-FLOW MIN-CUT THEOREM

THEOREM:

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

To prove this, we will prove 2 things:

LEMMA 1: value of max flow ≤ cost of min cut

Proof by picture!

LEMMA 2: value of max flow ≥ cost of min cut

Proof by algorithm (Ford-Fulkerson), which incrementally builds a flow f using a "residual graph" G_f.

MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow $\leq$ the cost of a min cut

MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow ≤ the cost of a min cut

Proof sketch:

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 value ≤ min cut cost.

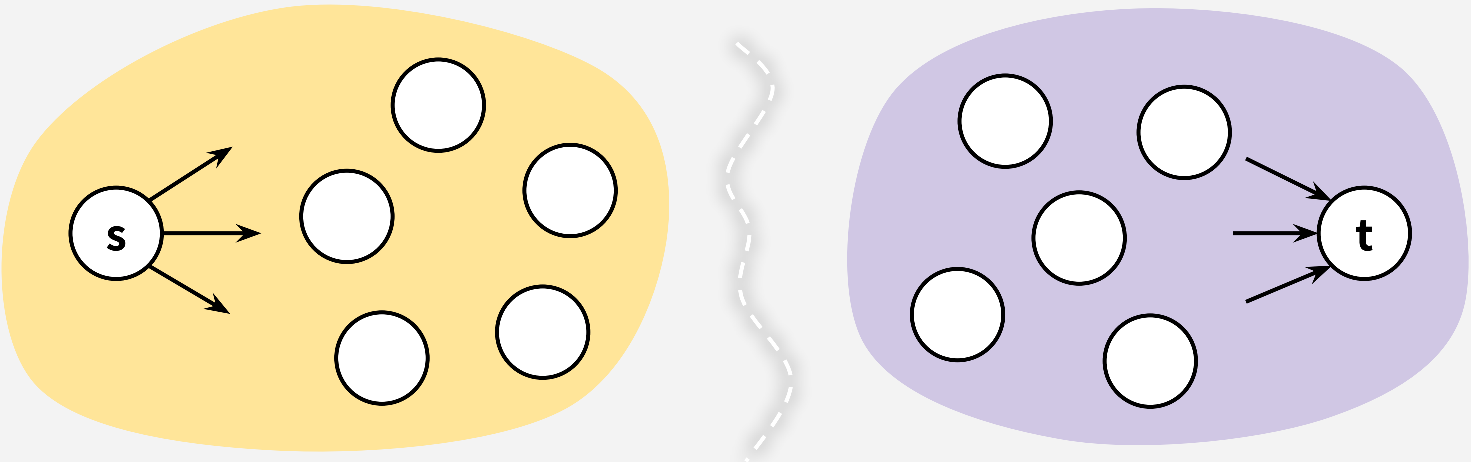
MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow ≤ the cost of a min cut

Proof sketch:

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 value ≤ min cut cost.

Proof by picture:



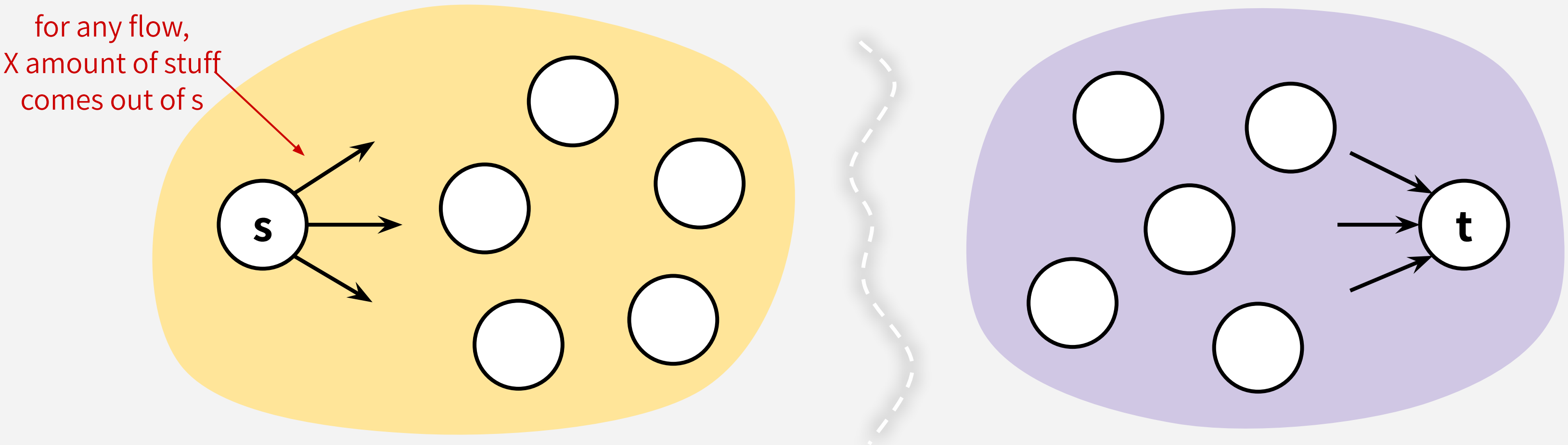
MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow ≤ the cost of a min cut

Proof sketch:

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 value ≤ min cut cost.

Proof by picture:



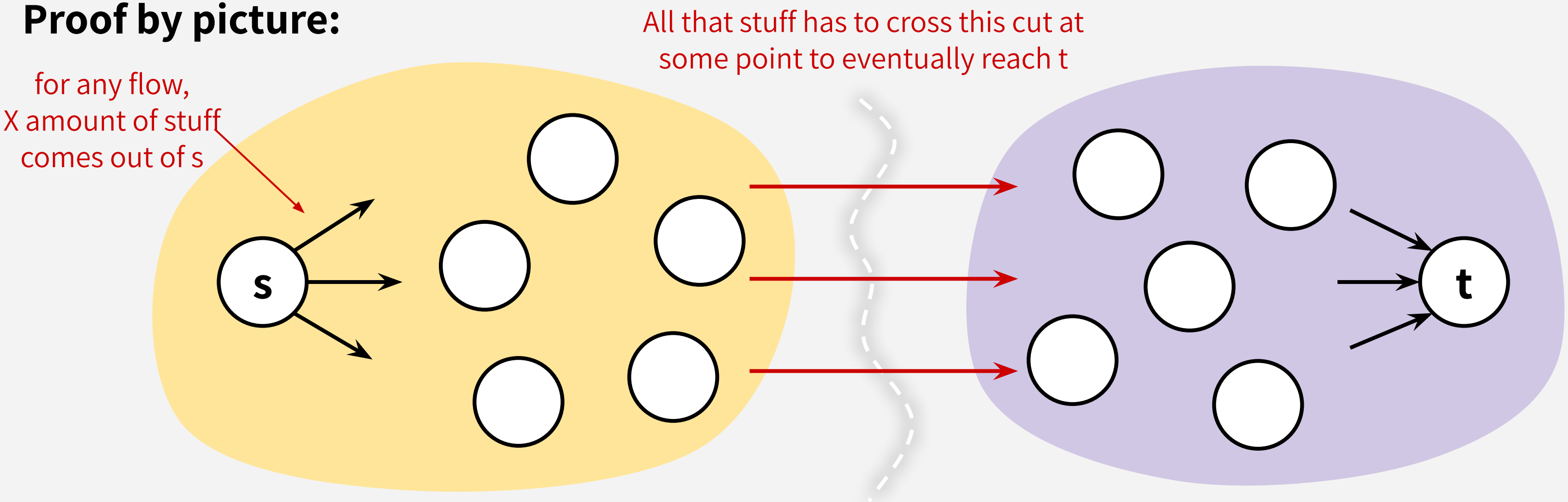
MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow ≤ the cost of a min cut

Proof sketch:

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 value ≤ min cut cost.

Proof by picture:



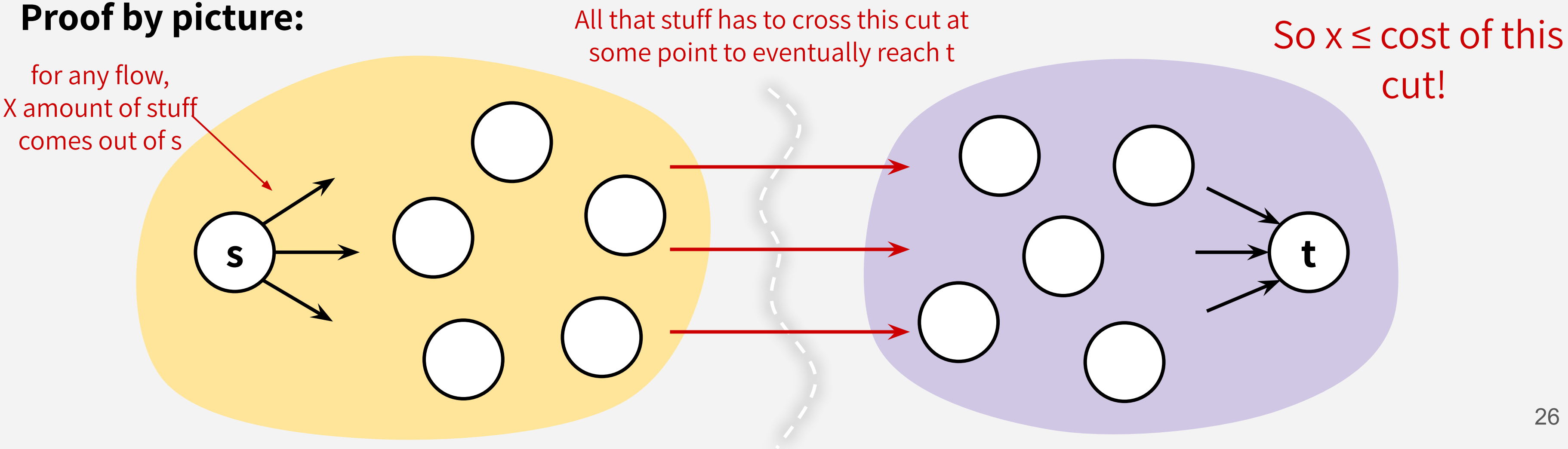
MAX-FLOW MIN-CUT THEOREM

LEMMA 1: the value of a max flow ≤ the cost of a min cut

Proof sketch:

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 value ≤ min cut cost.

Proof by picture:



MAX-FLOW MIN-CUT THEOREM

THEOREM:

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

To prove this, we will prove 2 things:

✓ **LEMMA 1:** value of max flow ≤ cost of min cut

Proof by picture!

LEMMA 2: value of max flow ≥ cost of min cut

Proof by algorithm (Ford-Fulkerson), which incrementally builds a flow f using a "residual graph" G_f.

سوال؟

الگوریتم فورد-فالکرسون

FORD-FULKERSON

FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

start with flow
⬇
find opportunity to throw water around
⬇
get better flow

FORD-FULKERSON

FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

start with flow
⬇
find opportunity to throw water around
⬇
get better flow

We'll define what a residual graph is. This will make sense in a bit, but here's a comment:

In my head, I like to call this an "opportunity graph"! I have the following story in mind: Your friend hands you some flow f, and you're tasked with finding new ways to throw water from s to t. To do so, you construct an "opportunity graph" that records all the available remaining opportunities you have to throw water around. If you find a new path of water-throwing in your opportunity graph, then "add" that path to your friend's flow f, and you've improved their flow!

FORD-FULKERSON: RESIDUAL GRAPH

FORD-FULKERSON(G, s, t):

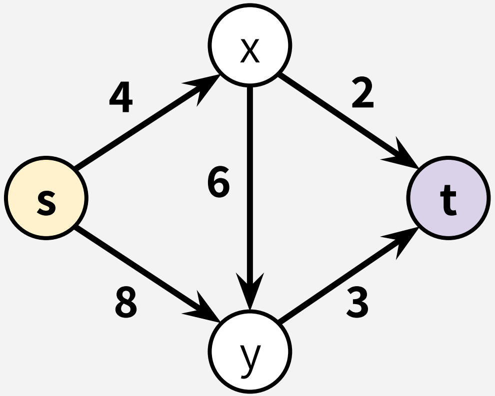
1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

FORD-FULKERSON: RESIDUAL GRAPH

FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G

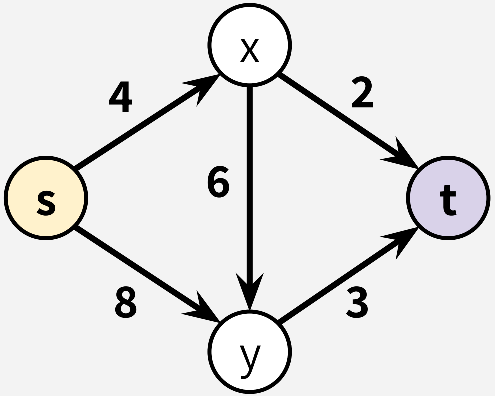


FORD-FULKERSON: RESIDUAL GRAPH

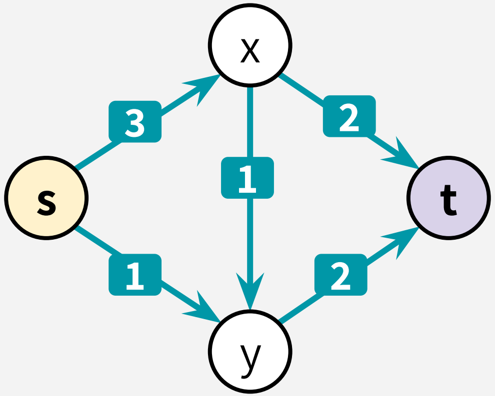
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G



SOME FLOW f

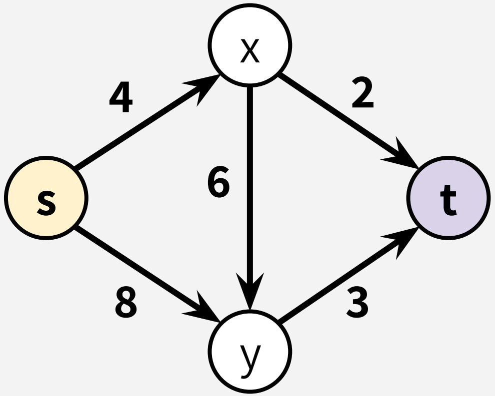


FORD-FULKERSON: RESIDUAL GRAPH

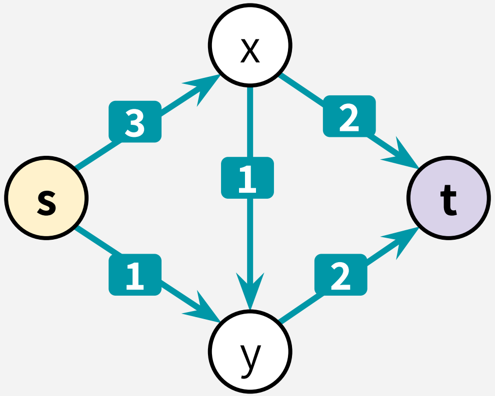
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G

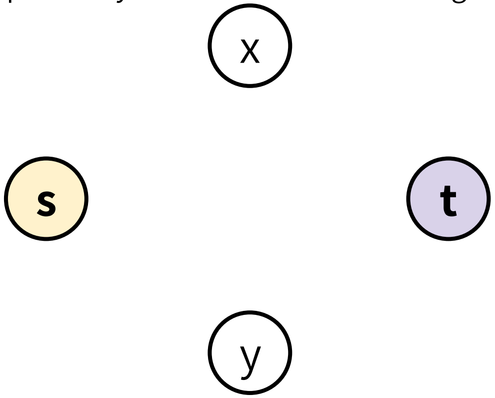


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)

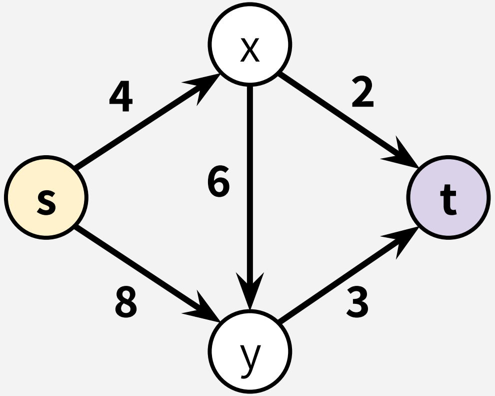


FORD-FULKERSON: RESIDUAL GRAPH

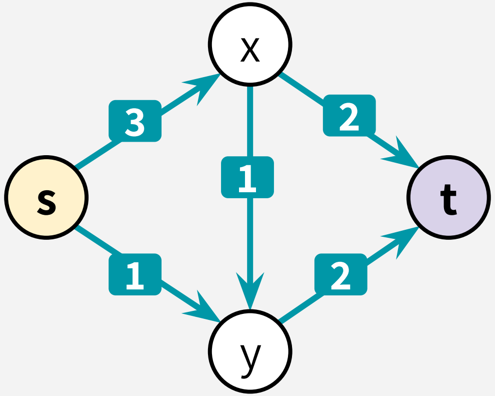
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G

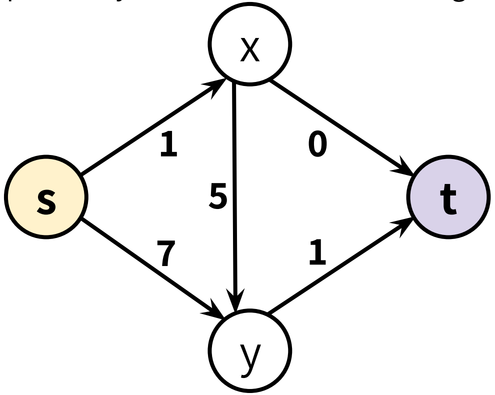


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



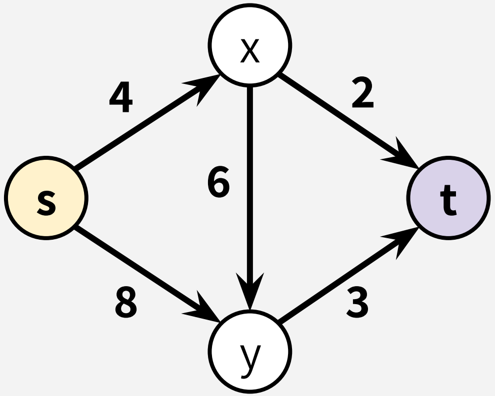
→ **"FORWARD EDGES"**
unused capacities in the original graph

FORD-FULKERSON: RESIDUAL GRAPH

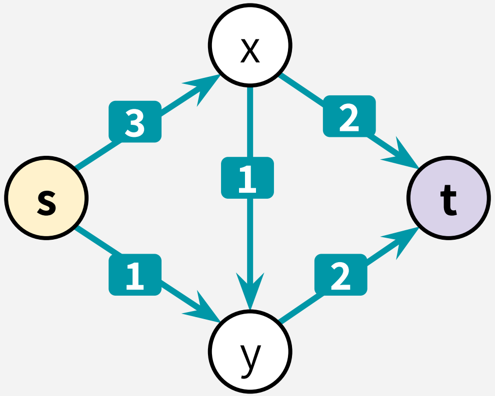
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G

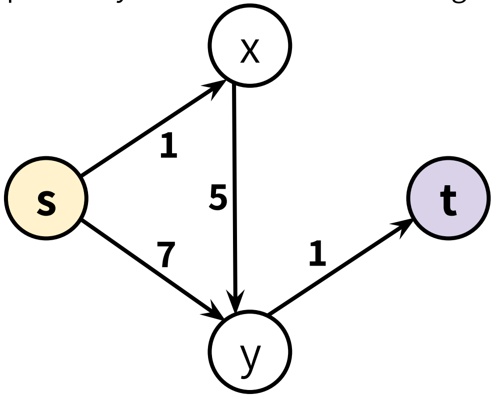


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

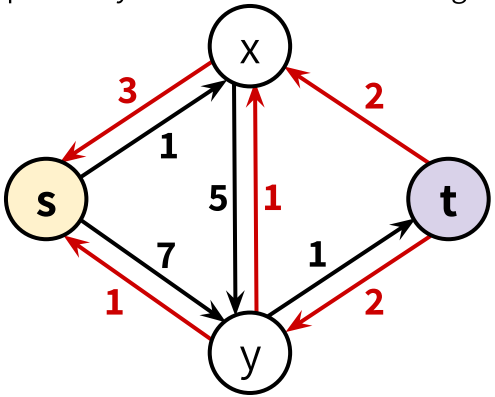
FORD-FULKERSON: RESIDUAL GRAPH

FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
    - if there is a path, update the flow f, and go back to step 2
    - if there isn't a path, then f is the max flow!

RESIDUAL GRAPH G_f

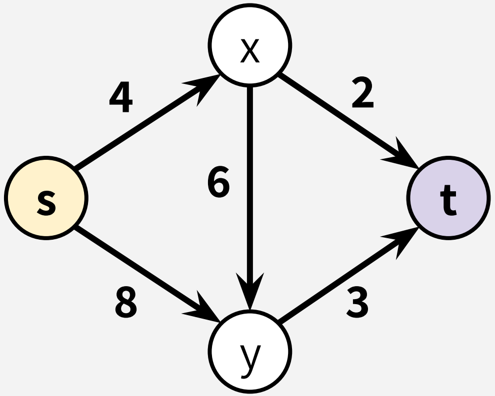
(opportunity-to-throw-water-around graph)



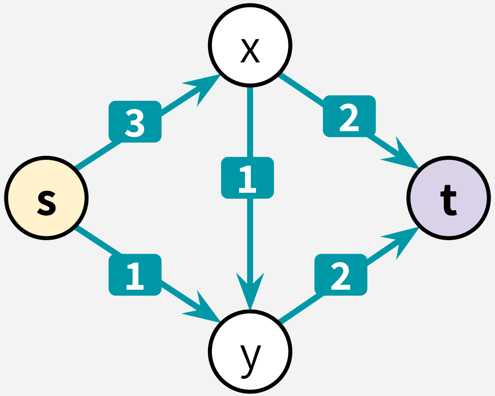
→ **"FORWARD EDGES"**
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

→ **"BACKWARD EDGES"**
capacities f already used, but backwards!

ORIGINAL GRAPH G



SOME FLOW f

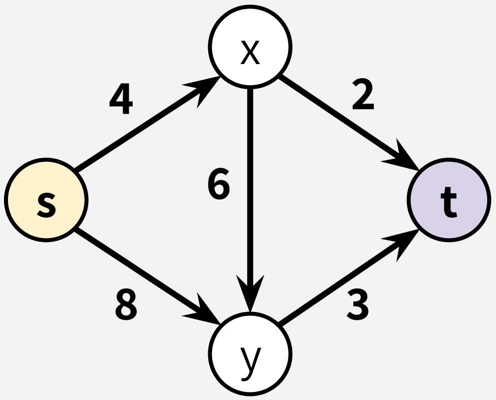


FORD-FULKERSON: RESIDUAL GRAPH

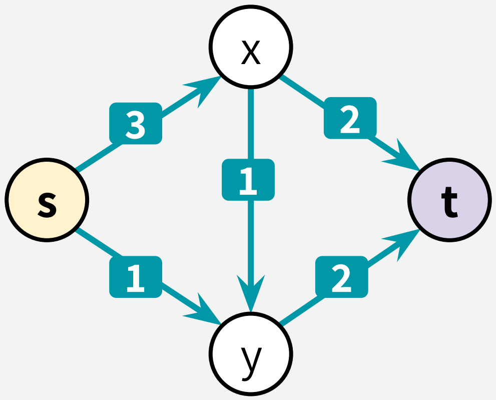
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t
   - if there is a path, update the flow f, and go back to step 2
   - if there isn't a path, then f is the max flow!

ORIGINAL GRAPH G

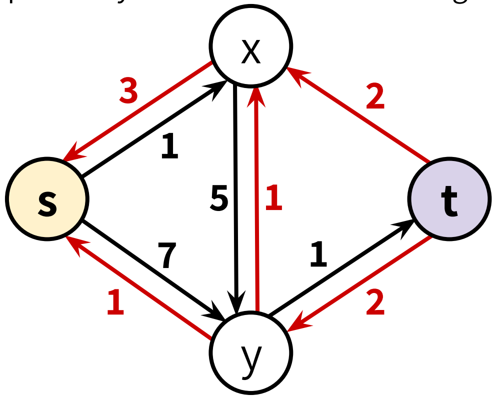


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

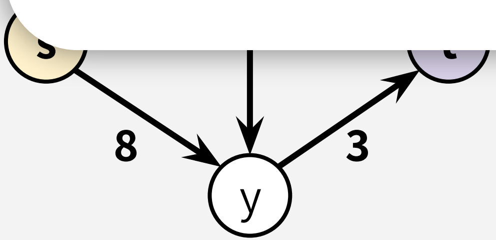
FORD-FULKERSON: RESIDUAL GRAPH

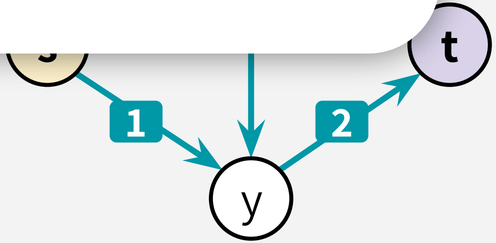
FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path in G_f from s to t

If we can find a s–t path in G_f,
then we've found an **augmenting path**.

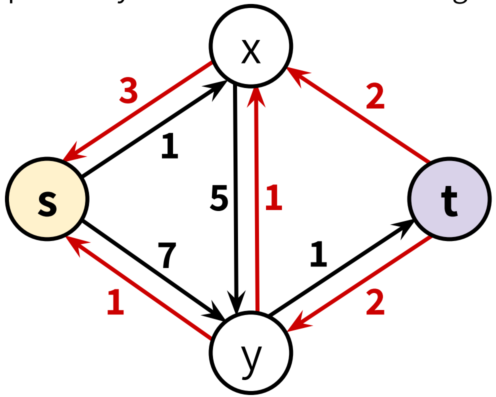
An augmenting path represents a way to improve our flow (we just "add" the path to our old flow!)





RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)

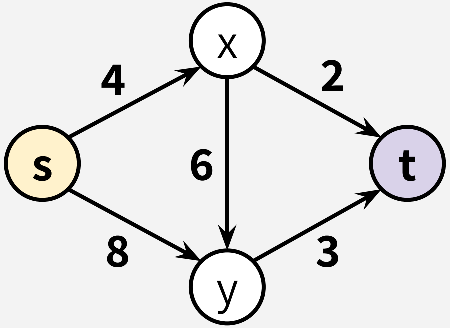


→ **"FORWARD EDGES"**
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

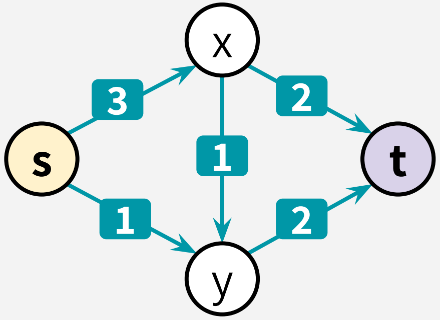
→ **"BACKWARD EDGES"**
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

AUGMENTING PATH EXAMPLE 1

ORIGINAL GRAPH G

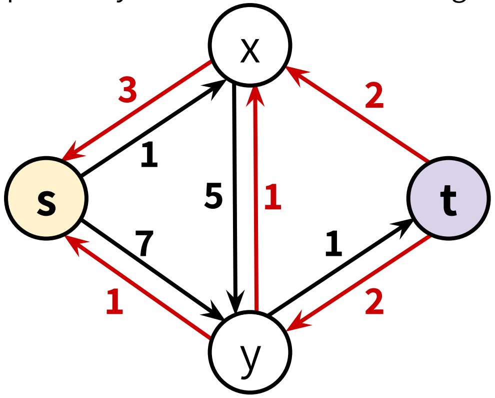


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



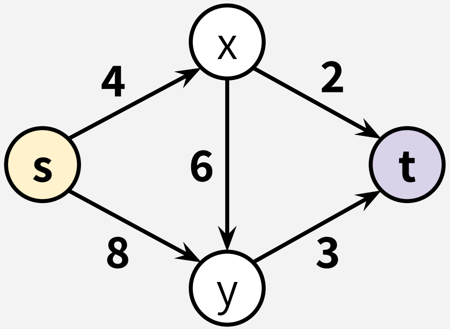
"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

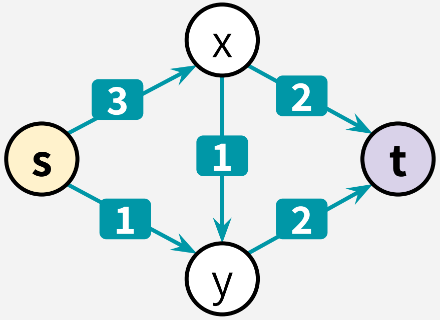
Let's find an augmenting path in G_f

AUGMENTING PATH EXAMPLE 1

ORIGINAL GRAPH G

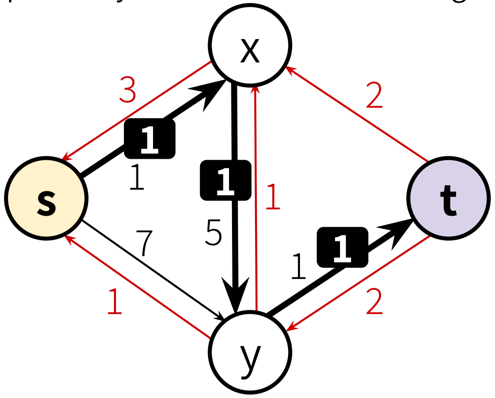


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

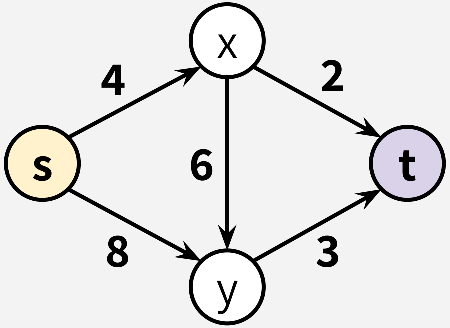
"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

Let's find an augmenting path in G_f

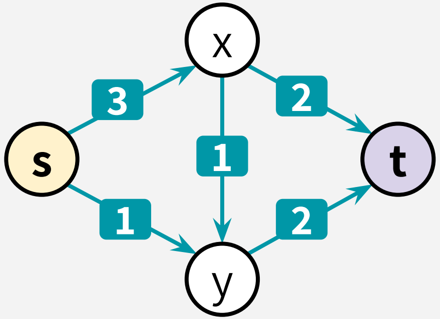
Here's one!
(there may be multiple, but just pick one)

AUGMENTING PATH EXAMPLE 1

ORIGINAL GRAPH G

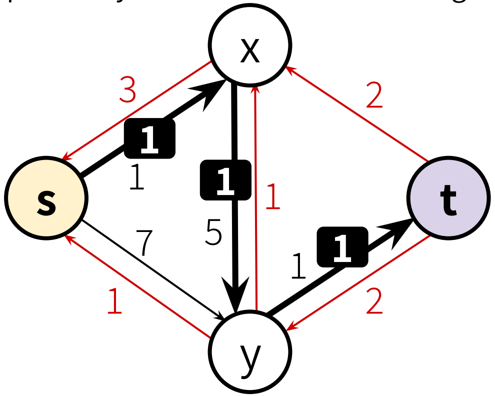


SOME FLOW f



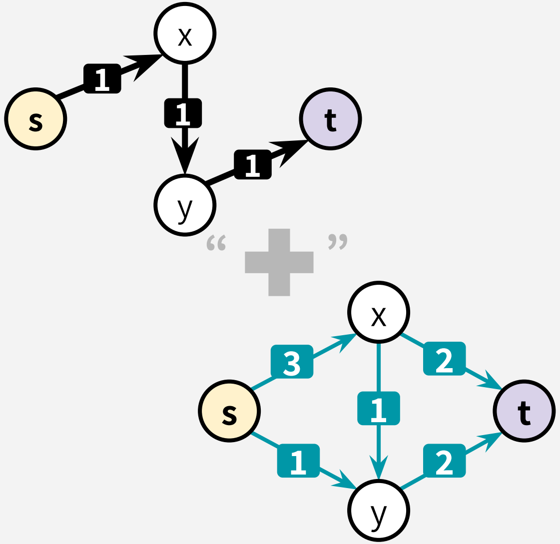
RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



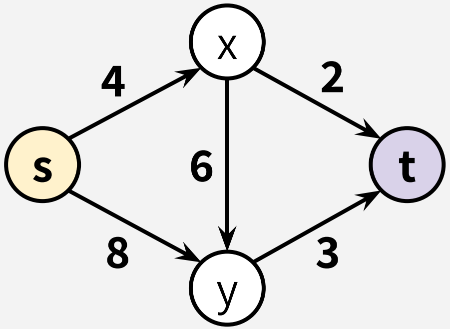
"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

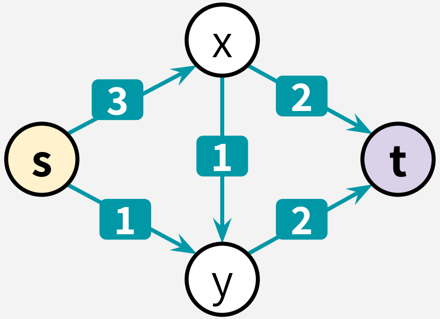


AUGMENTING PATH EXAMPLE 1

ORIGINAL GRAPH G

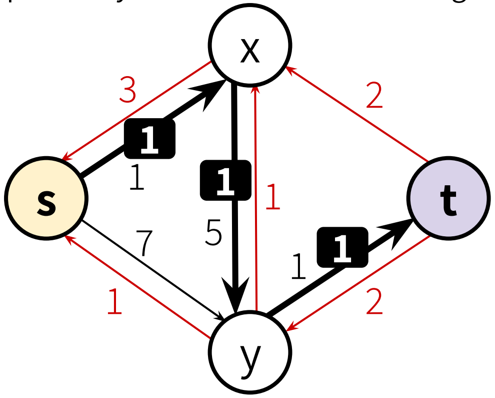


SOME FLOW f



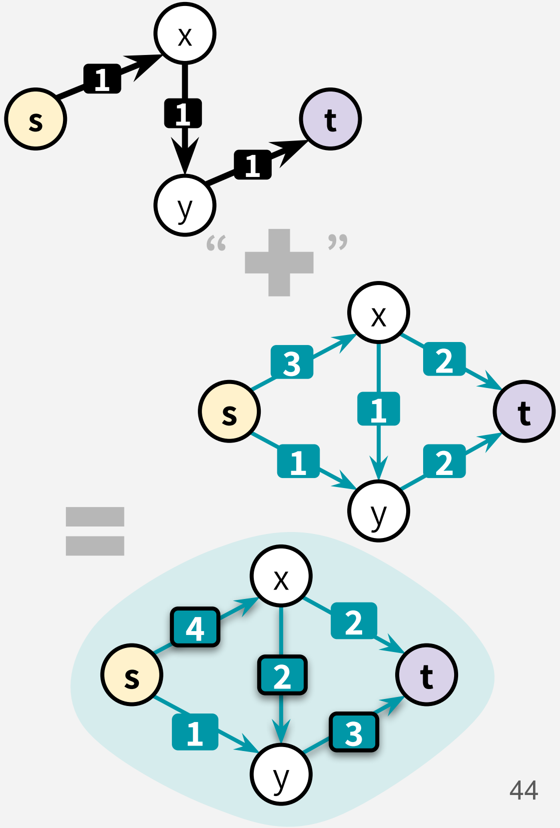
RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



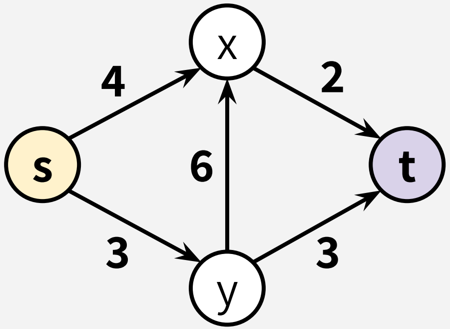
“FORWARD EDGES”
unused capacities in the original graph
(you can throw water that your friend’s flow didn’t use up)

“BACKWARD EDGES”
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

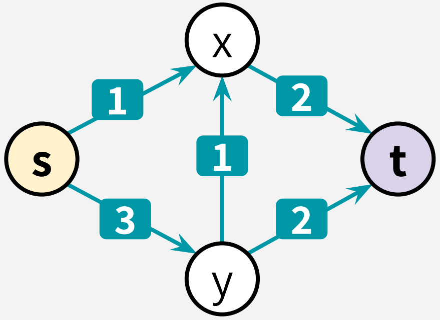


AUGMENTING PATH EXAMPLE 2

ORIGINAL GRAPH G

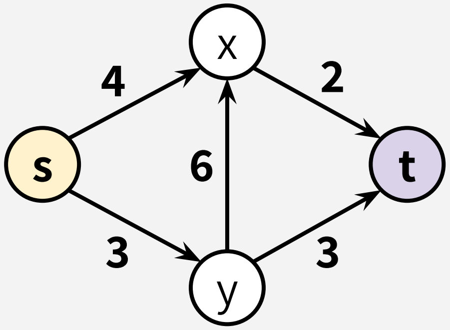


SOME FLOW f

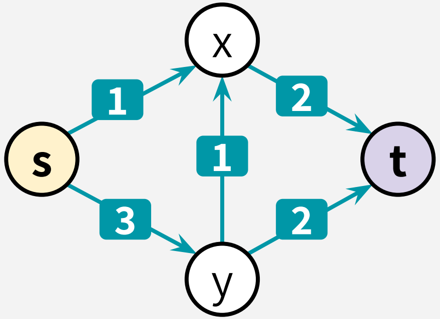


AUGMENTING PATH EXAMPLE 2

ORIGINAL GRAPH G

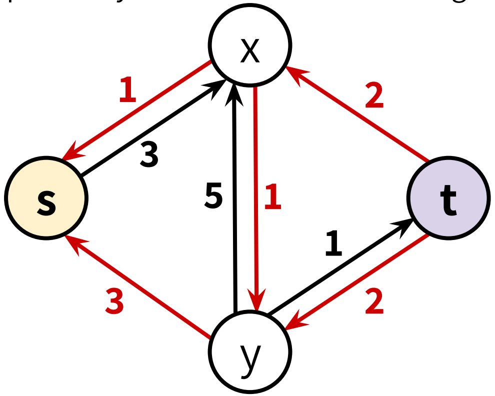


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



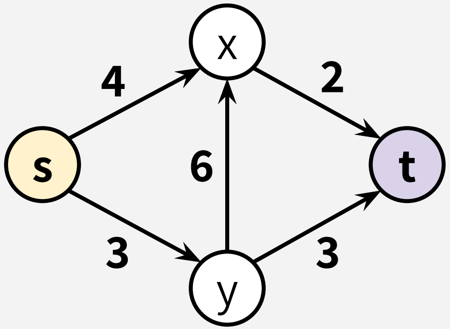
"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

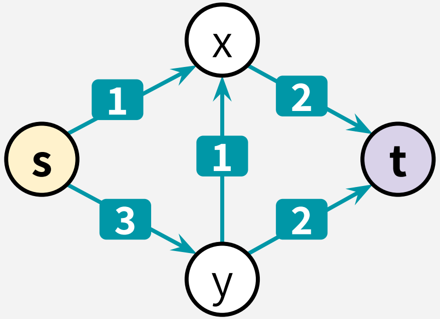
Let's find an augmenting path in G_f

AUGMENTING PATH EXAMPLE 2

ORIGINAL GRAPH G

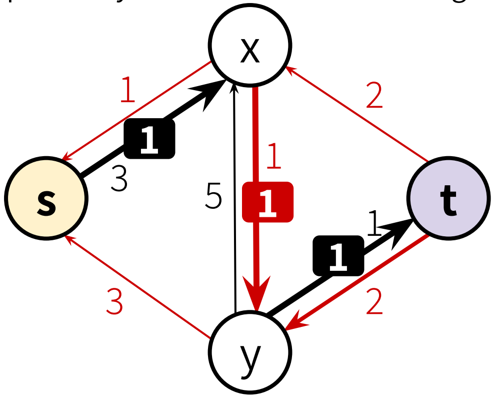


SOME FLOW f



RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



→ **"FORWARD EDGES"**
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

→ **"BACKWARD EDGES"**
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

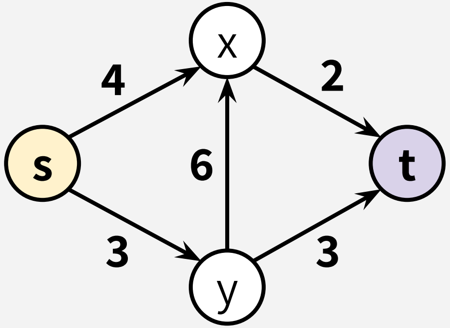
Let's find an augmenting path in G_f

Here's one!

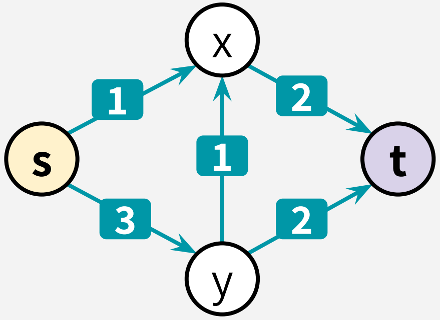
Note that it takes a backwards edge! This indicates that you should probably **"undo"** something in your original flow (in this case, notice that the flow of 1 from y → x just looks like a bad decision…). Having these backwards edges in our residual graph gives us a chance to undo these bad decisions!

AUGMENTING PATH EXAMPLE 2

ORIGINAL GRAPH G

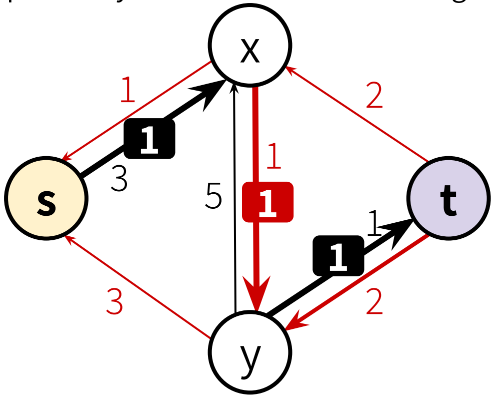


SOME FLOW f



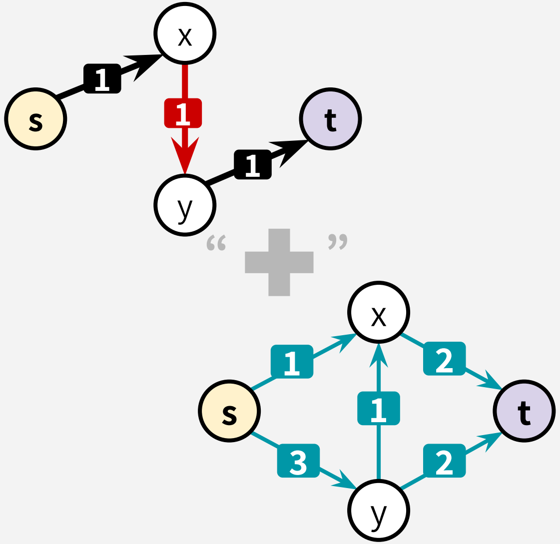
RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



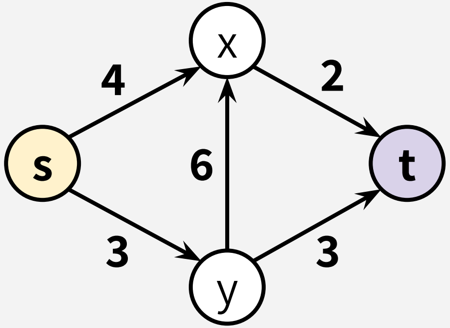
"FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

"BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)

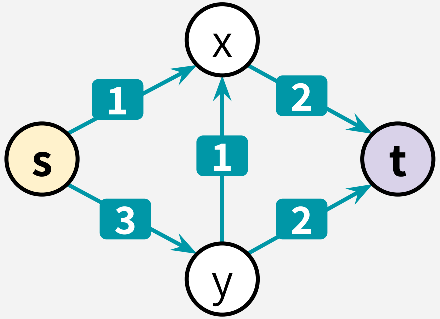


AUGMENTING PATH EXAMPLE 2

ORIGINAL GRAPH G

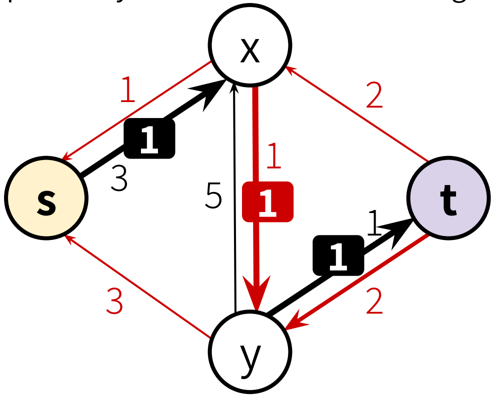


SOME FLOW f



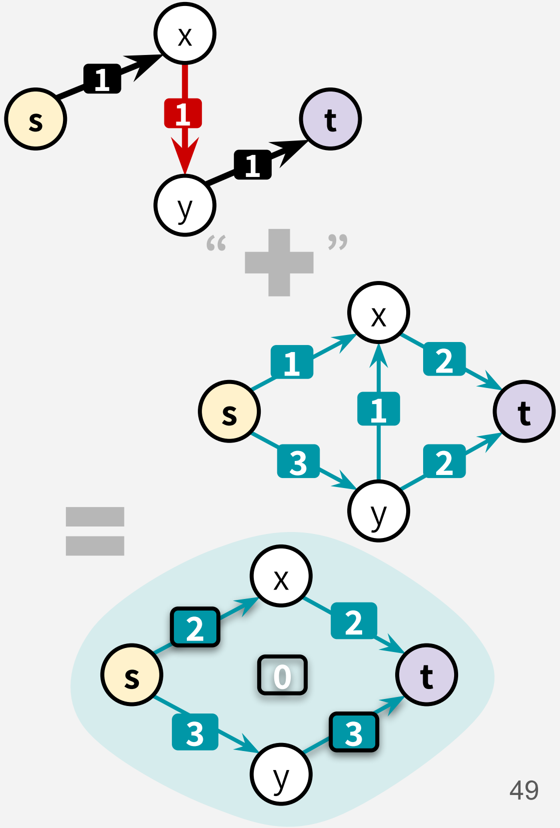
RESIDUAL GRAPH G_f

(opportunity-to-throw-water-around graph)



→ "FORWARD EDGES"
unused capacities in the original graph
(you can throw water that your friend's flow didn't use up)

→ "BACKWARD EDGES"
capacities f already used, but backwards!
(if your friend threw X amount of water one way, you have the opportunity to throw back their water in the reverse direction)



AUGMENTING PATH PROCEDURE

UPDATE_FLOW(path P in G_f , flow f):

- x = min weight on any edge in P from G_f
- for (u, v) in P:
  - if (u, v) in E: $f_{new}(u, v) = f(u, v) + x$
  - if (v, u) in E: $f_{new}(u, v) = f(u, v) - x$
- return f_{new}

start with flow
⬇
find opportunity to throw water around
⬇
get better flow

Note: you should convince yourself that increasing flow along an augmenting path will result in a **larger** & **legitimate** flow!

FORD FULKERSON

FORD-FULKERSON(G, s, t):

1. Start with arbitrary flow f (let's say flow of 0)
2. Construct residual graph G_f
3. Check if there's a path P in G_f from s to t
   - if there is a path P, f = **UPDATE_FLOW**(P, f), & go back to step 2
   - if there isn't a path, then f is the max flow!

start with flow

⬇

find opportunity to throw water around

⬇

get better flow

MAX-FLOW MIN-CUT THEOREM

THEOREM:

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

To prove this, we will prove 2 things:

✓ **LEMMA 1:** value of max flow ≤ cost of min cut

Proof by picture!

We still need to finish proving LEMMA 2, and we'll use Ford-Fulkerson to do that...

LEMMA 2: value of max flow ≥ cost of min cut

Proof by algorithm (Ford-Fulkerson), which incrementally builds a flow f using a "residual graph" G_f.

سوال؟

اثبات درستی الگوریتم فورد- فالکرسون

MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

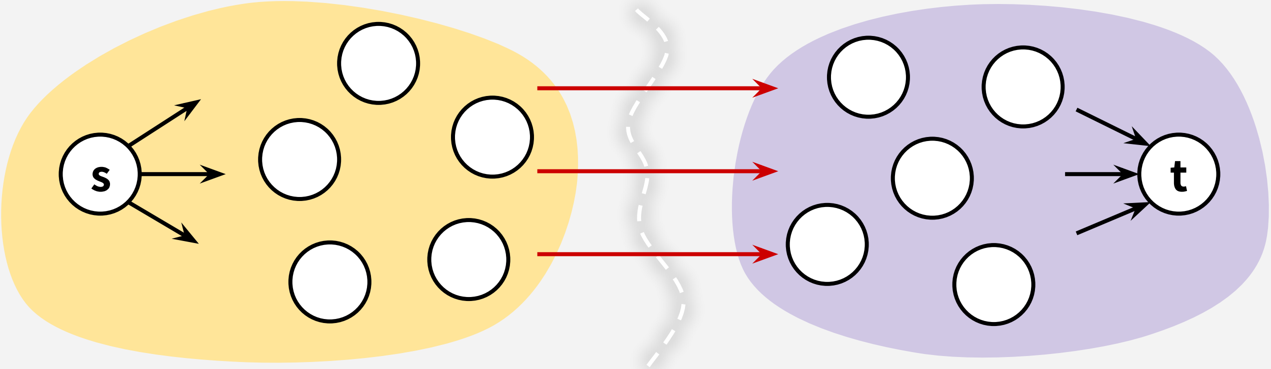
MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.



MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

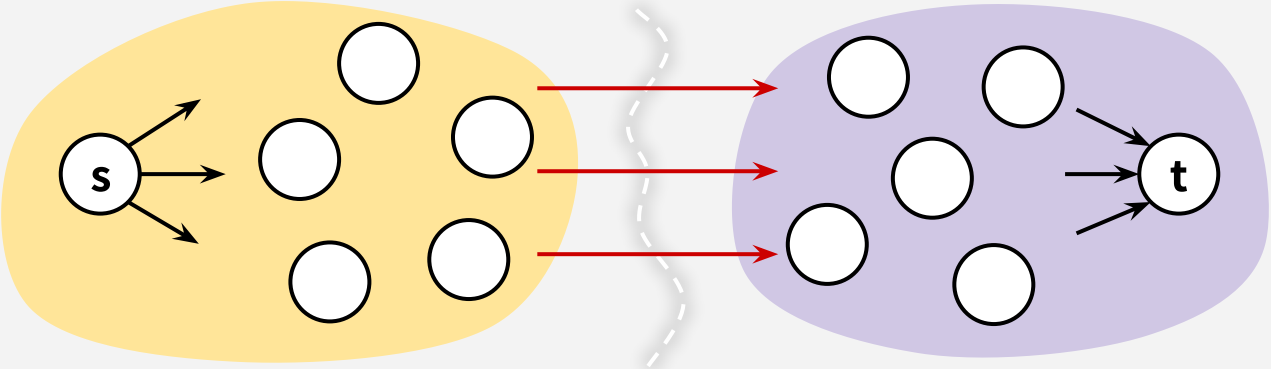
Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f

(if they existed in G_f, then s could still reach t)



MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f

(if they existed in G_f, then s could still reach t)

So it turns out that when Ford-Fulkerson stops, the current f must be a max flow:

f's flow value = cost of some cut

from above

MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**
The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f
(if they existed in G_f, then s could still reach t)

So it turns out that when Ford-Fulkerson stops, the current f must be a max flow:

f's flow value = cost of some cut ≥ cost of min cut ≥ max flow value

from above

definition of minimum cut

Lemma 1

MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f

(if they existed in G_f, then s could still reach t)

We haven't proved the inequality in the Lemma yet, but it's nice to know Ford-Fulkerson does succeed in finding a max flow!!

So it turns out that when Ford-Fulkerson stops, the current f must be a max flow:

f's flow value = cost of some cut ≥ cost of min cut ≥ max flow value

from above

definition of minimum cut

Lemma 1

MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f

(if they existed in G_f, then s could still reach t)

But also, this means that:

f's flow value = cost of some cut

from above

MAX-FLOW MIN-CUT THEOREM

LEMMA 2: the value of a max flow ≥ the cost of a min cut

Proof: We'll first prove that if there is no augmenting path, our flow **f** is a max flow.

Consider the cut **{things reachable from s (in G_f)}**, **{things not reachable from s (in G_f)}**

The flow from s to t is *equal* to the cost of this cut.

The edges in the cut must be **full** because they don't exist in G_f

(if they existed in G_f, then s could still reach t)

But also, this means that:

max flow value ≥ f's flow value = cost of some cut ≥ cost of min cut

definition of maximum flow

from above

definition of minimum cut

MAX-FLOW MIN-CUT THEOREM

THEOREM:

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

To prove this, we will prove 2 things:

✓ **LEMMA 1:** value of max flow ≤ cost of min cut

Proof by picture!

✓ **LEMMA 2:** value of max flow ≥ cost of min cut

Proof by algorithm (Ford-Fulkerson), which incrementally builds a flow f using a "residual graph" G_f.
We basically thought about why Ford-Fulkerson works, and it led us to show that max flow ≥ min cut!

FORD FULKERSON: ~PSEUDOCODE

FORD-FULKERSON(G, s, t):

```
f = all zero flow
G_f = G
while t is reachable from s in G_f (e.g. use BFS):
    get an s–t path P in G_f
    f = INCREASE_FLOW(P, f)
    update G_f
return f
```

Using BFS results in what's called the EDMONDS-KARP Algorithm

start with flow
⬇
find opportunity to throw water around
⬇
get better flow

Runtime (using BFS to find augmenting paths): $O(nm^2)$

We will not prove this runtime in class! It's quite an involved proof, but if you're curious, the full is in the book!

سوال؟

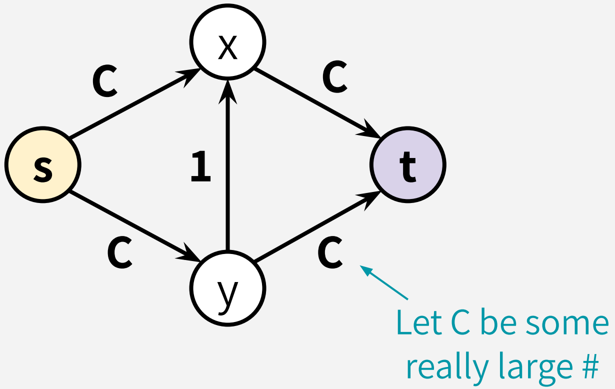
نکاتی درباره الگوریتم فورد-فالکرسون

FORD FULKERSON: SOME NOTES

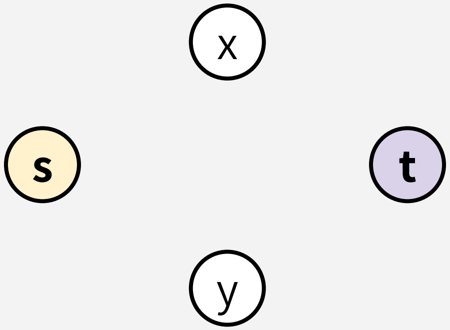
Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

For example, this would be a bad way to pick paths:

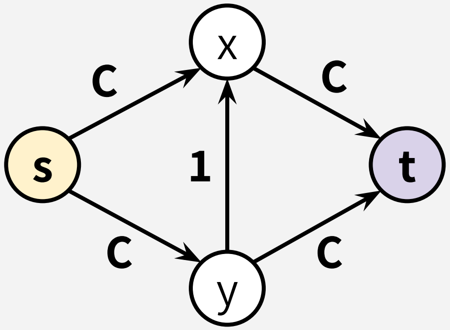
ORIGINAL GRAPH G



OUR FLOW f



RESIDUAL GRAPH G_f

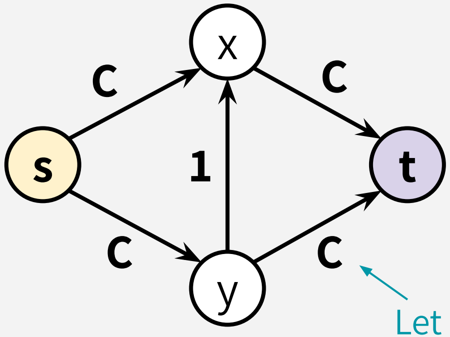


FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

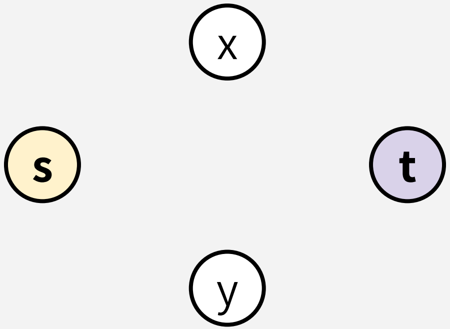
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G



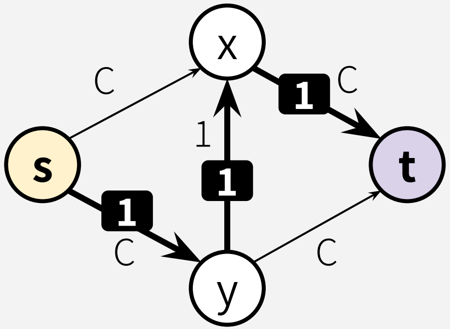
Let C be some
really large #

OUR FLOW f



(find augmented path)

RESIDUAL GRAPH G_f

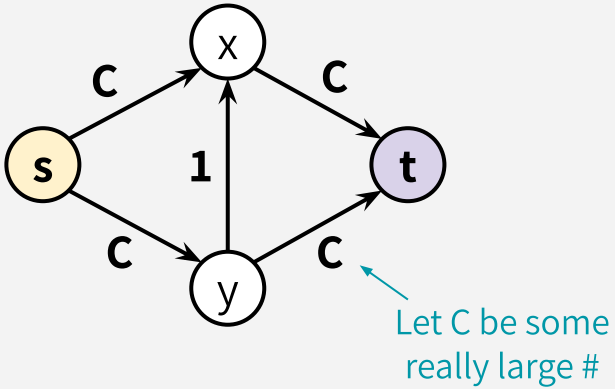


FORD FULKERSON: SOME NOTES

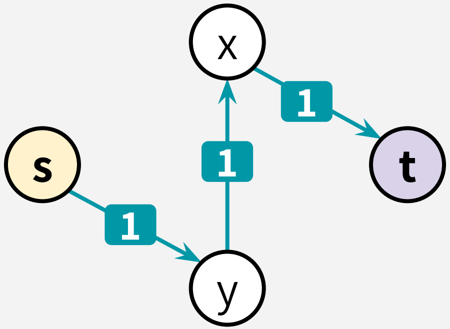
Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

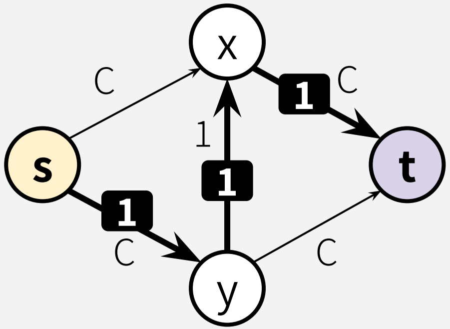


OUR FLOW f



(update flow)

RESIDUAL GRAPH G_f

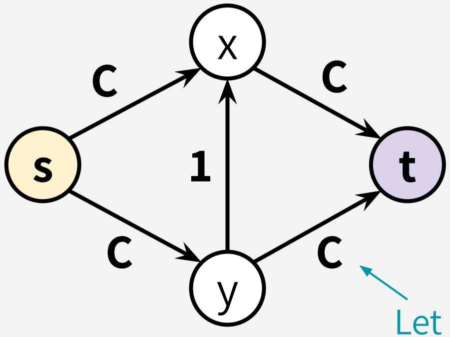


FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

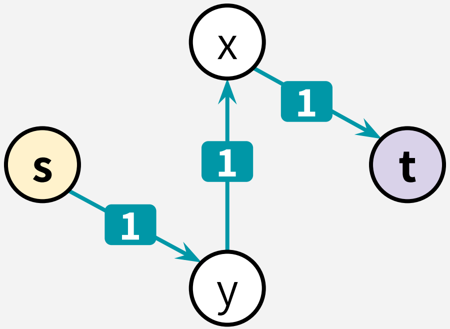
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G



Let C be some really large #

OUR FLOW f



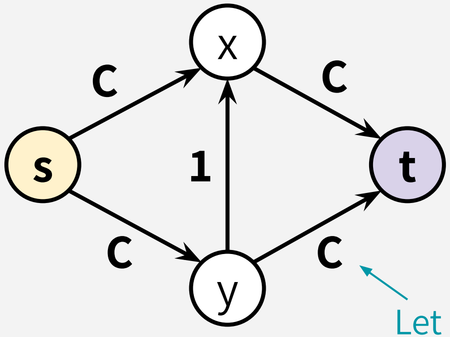
(update residual graph)

FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

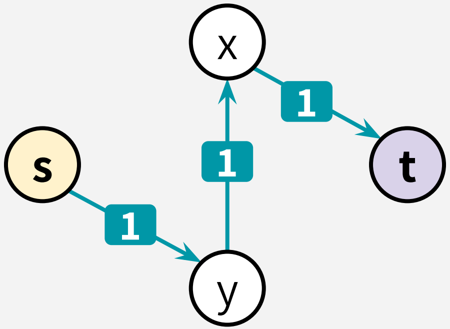
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

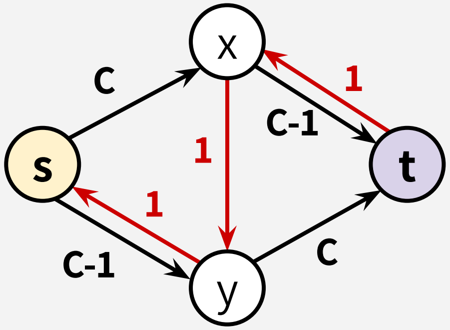


Let C be some really large #

OUR FLOW f



RESIDUAL GRAPH G_f



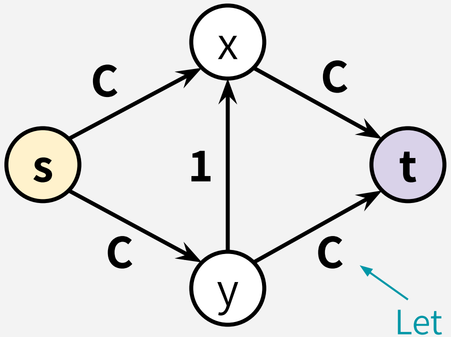
(update residual graph)

FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

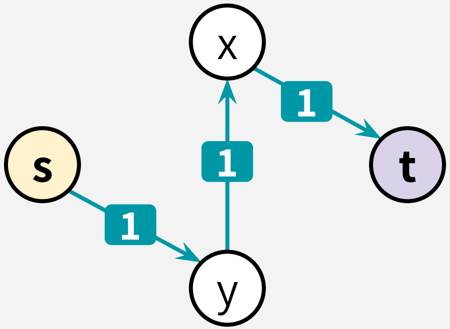
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

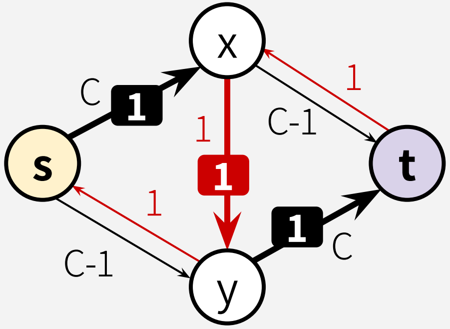


Let C be some really large #

OUR FLOW f



RESIDUAL GRAPH G_f

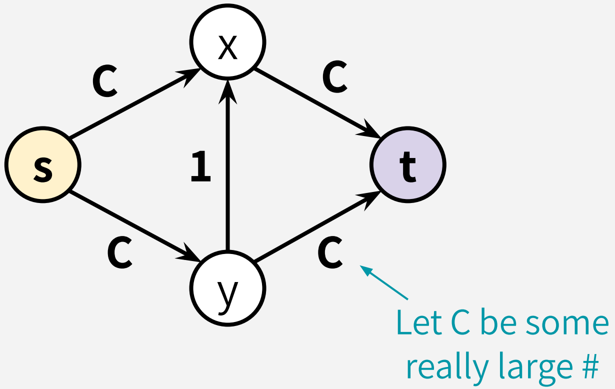


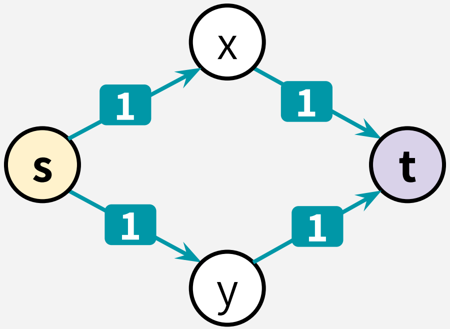
(find augmented path)

FORD FULKERSON: SOME NOTES

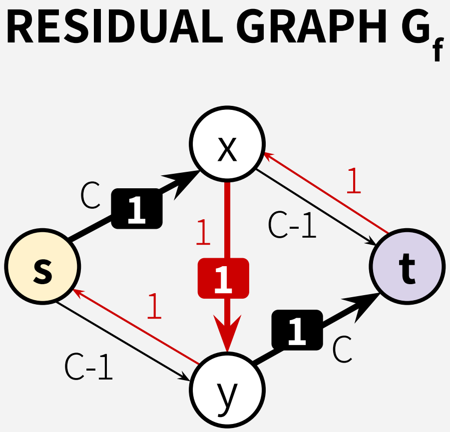
Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

For example, this would be a bad way to pick paths:





(update flow)

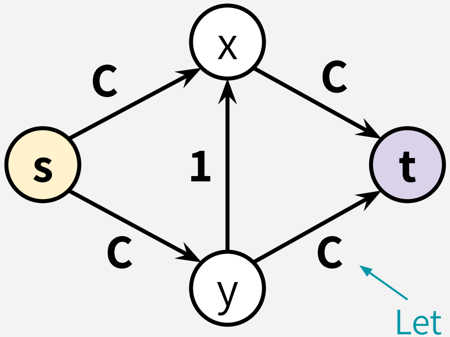


FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

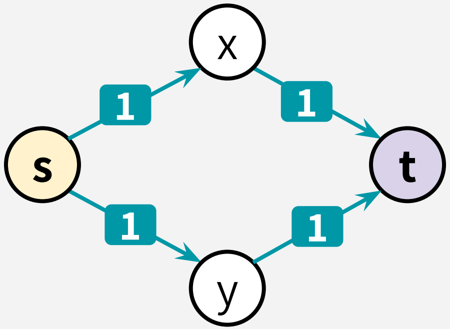
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

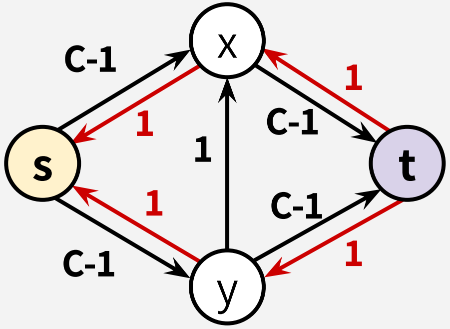


Let C be some really large #

OUR FLOW f



RESIDUAL GRAPH G_f



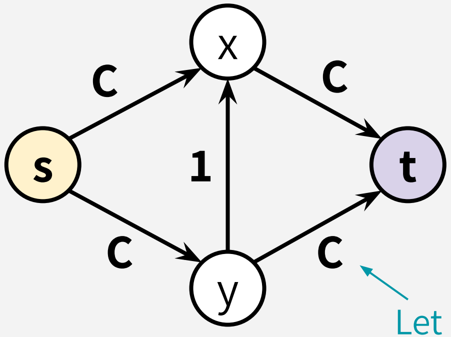
(update residual graph)

FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

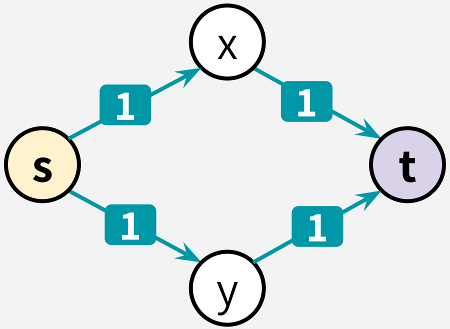
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

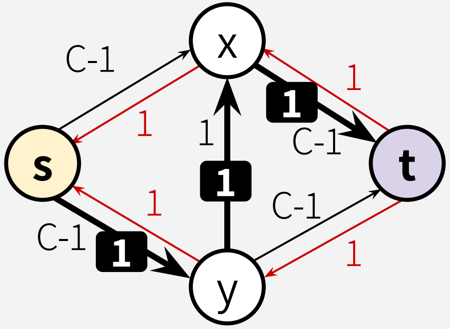


Let C be some really large #

OUR FLOW f



RESIDUAL GRAPH G_f



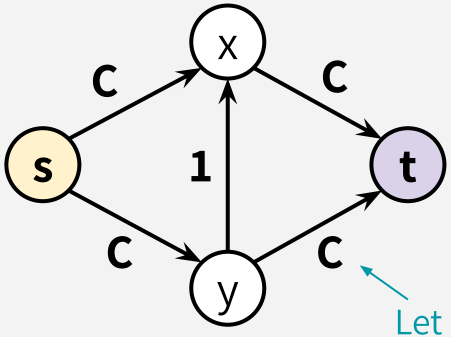
(find augmented path)

FORD FULKERSON: SOME NOTES

Not all augmenting path-finding procedures are created equal:
We need to be careful about *how* we select an augmenting path.

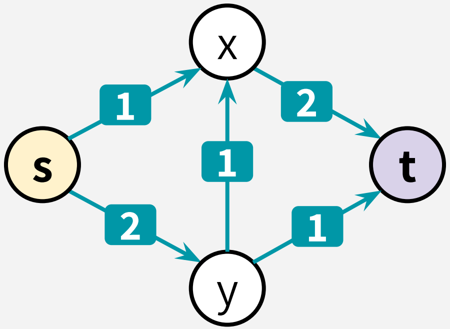
For example, this would be a bad way to pick paths:

ORIGINAL GRAPH G

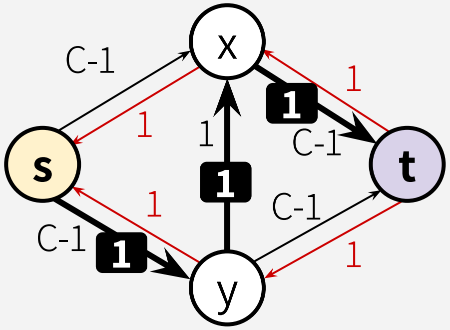


Let C be some really large #

OUR FLOW f



RESIDUAL GRAPH G_f



(update flow)

FORD FULKERSON: SOME NOTES

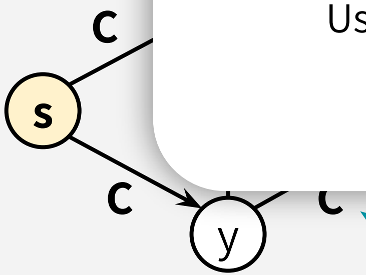
Not all augmenting path-finding procedures are created equal:

If we're not thoughtful about how we select our augmenting path, then this could go on for a while…

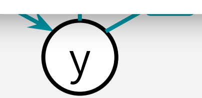
The algorithm will ultimately be correct, but the way in which we discover augmenting paths determines how efficient our algorithm is!

Using BFS is one way to be "smart" about our path discovery!
There are other ways too! (see the book for details)

ORIGI

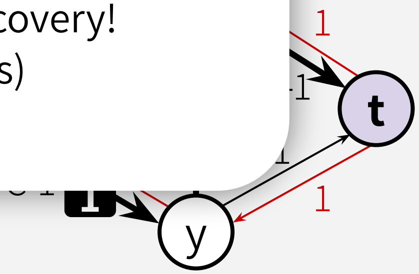


Let C be some really large #



(update flow)

PH G_f



FORD FULKERSON: SOME NOTES

Also, if the capacities of the input graph are all integers, then the value of any max flow is also an integer!

When we update flows in Ford-Fulkerson, we're only ever adding or subtracting integers! So, since we started with a flow of value 0 (which is an integer), our flow will only ever have an integer value.

s–t MIN CUT & MAX FLOW: RECAP

What have we learned?

Finding the Max s–t flow is equal to finding the min s–t cut!

So the USSR and USA were trying to solve the same problem…

Ford-Fulkerson is a method for finding the max-flow/min-cut!

Use augmenting paths to find the max-flow. In the final residual graph, the cut that separates nodes reachable by s and nodes not reachable by s is the min-cut

There are different ways to discover augmenting paths!

Edmonds-Karp uses BFS to discover an augmenting path. There are other ways!

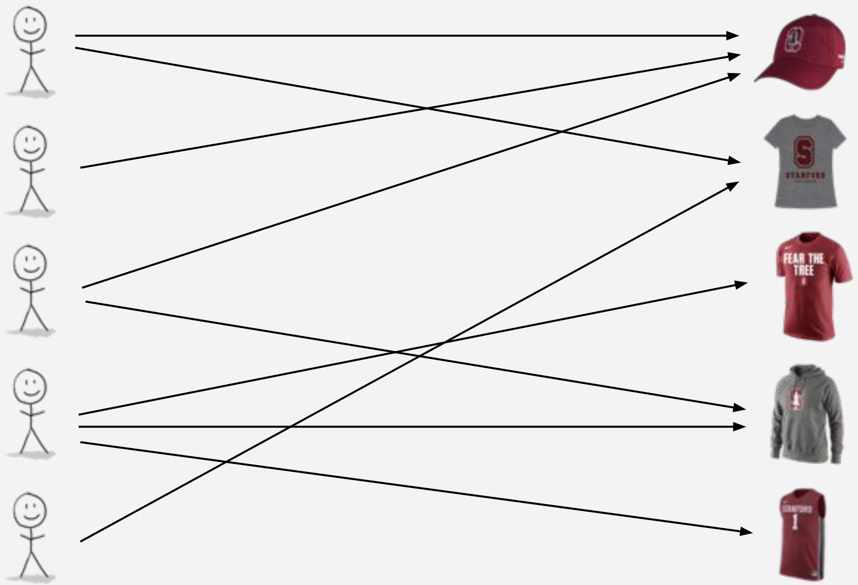
سوال؟

کاربردهای شاره بیشینه

AN APPLICATION: BIPARTITE MATCHING

Suppose we have a group of students and some items. Each student only would want certain items (depending on fit, style, etc.), and I only have one of each item.

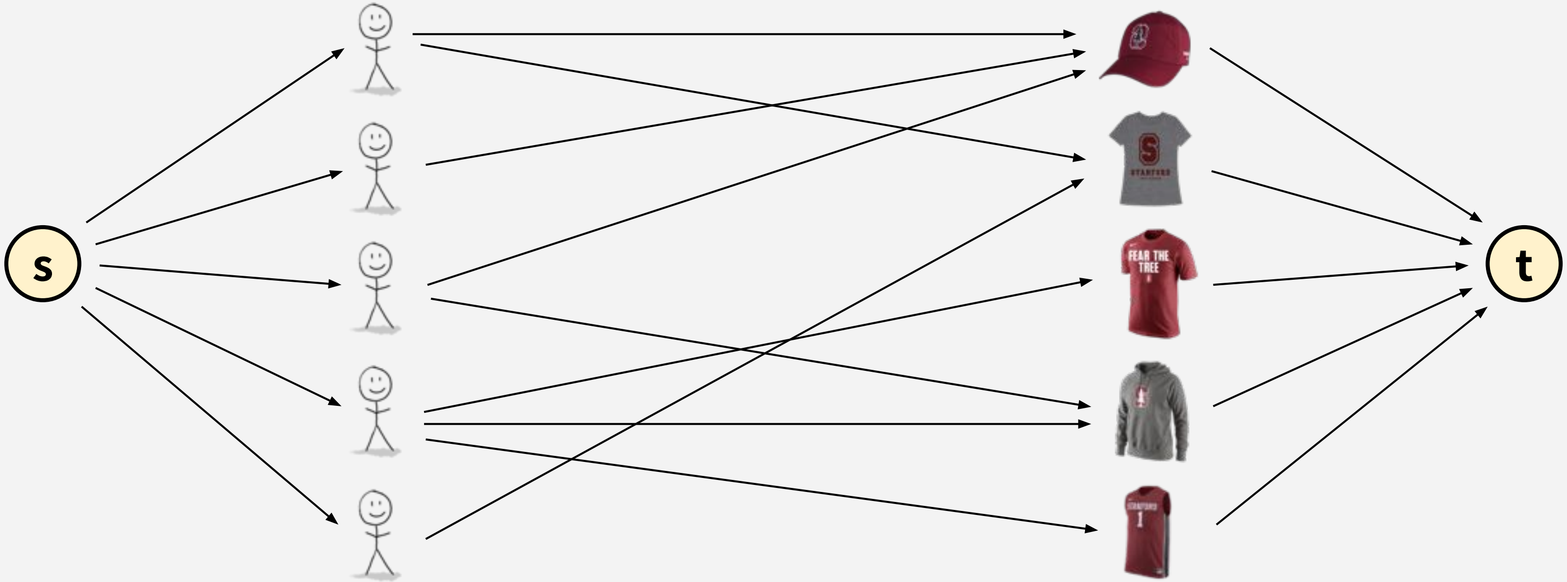
How can we make as many students as possible happy?



AN APPLICATION: BIPARTITE MATCHING

Turn this into a Max-Flow problem!

Add a source node s and a sink node t. Give all edges a capacity of 1.

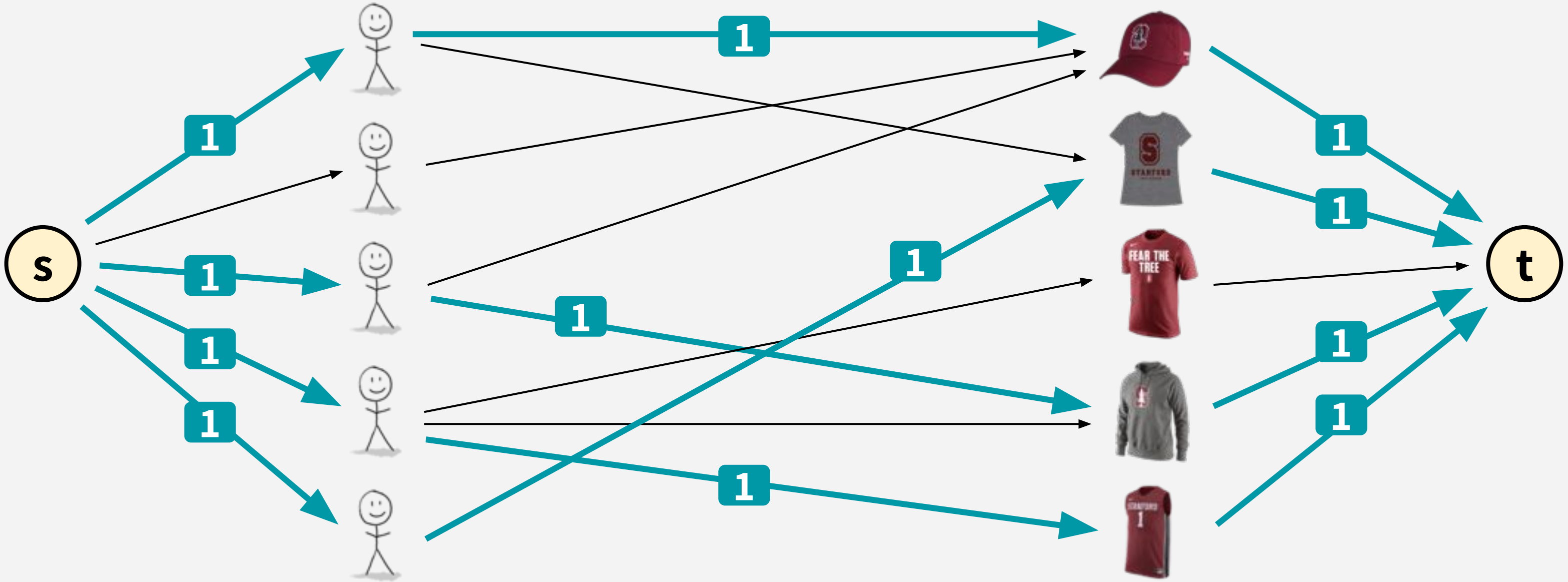


AN APPLICATION: BIPARTITE MATCHING

Turn this into a Max-Flow problem!

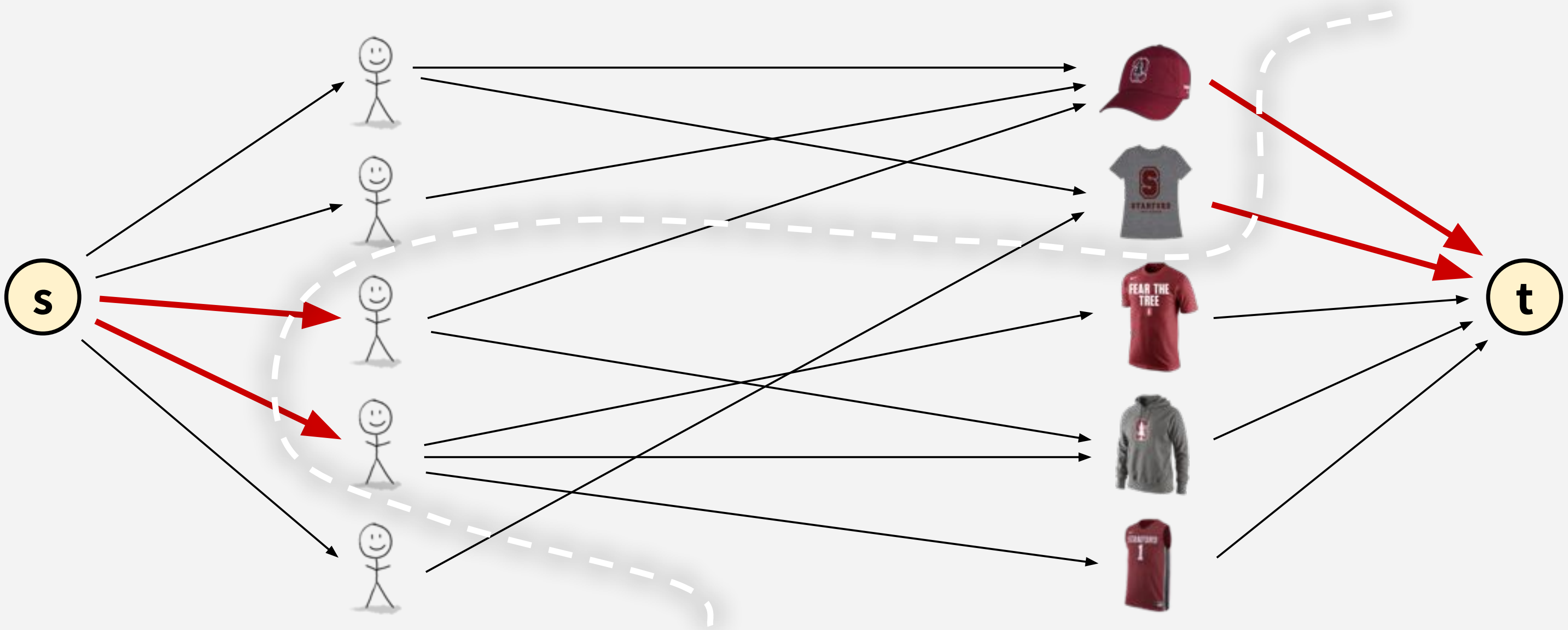
Add a source node s and a sink node t. Give all edges a capacity of 1.

Any student → item edge that is filled up denotes an assignment!



AN APPLICATION: BIPARTITE MATCHING

Also, for those curious, this is the min-cut of this graph!
It has cost 4 (same as the max-flow value).
(Remember, only edges that cross from the s-side to the t-side count towards the cost)



AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

Students each want different amounts of ice cream scoops.
Each student has certain ice cream flavor preferences.
Each ice cream tub has a certain number of scoops available.

Goal: provide as many scoops of ice cream as possible!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

Students each want different amounts of ice cream scoops.
Each student has certain ice cream flavor preferences.
Each ice cream tub has a certain number of scoops available.

Goal: provide as many scoops of ice cream as possible!

Create a source node s, and a sink node t.
Source → student edges have capacity representing the # of ice cream scoops that student wants.
Ice cream → sink edges have capacity representing the # of ice cream scoops available in that tub.
Student → ice cream edges have infinity capacity!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

Students each want different amounts of ice cream scoops.
Each student has certain ice cream flavor preferences.
Each ice cream tub has a certain number of scoops available.
Each student can eat a maximum of 3 scoops out of any given ice cream tub.

Goal: provide as many scoops of ice cream as possible!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

Students each want different amounts of ice cream scoops.
Each student has certain ice cream flavor preferences.
Each ice cream tub has a certain number of scoops available.
Each student can eat a maximum of 3 scoops out of any given ice cream tub.

Goal: provide as many scoops of ice cream as possible!

Create a source node s, and a sink node t.
Source → student edges have capacity representing the # of ice cream scoops that student wants.
Ice cream → sink edges have capacity representing the # of ice cream scoops available in that tub.
Student → ice cream edges have capacity 3!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

A group of housemates have bought different amounts of house-groceries over a few months, and now they want to split the costs evenly.

Goal: figure out what payments should happen to make costs even!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

A group of housemates have bought different amounts of house-groceries over a few months, and now they want to split the costs evenly.

Goal: figure out what payments should happen to make costs even!

For each person, compute how much they owe/are owed.
Two groups = shouldPay people & shouldn't Pay people.
Create a source node s, and a sink node t.
Source → shouldPay edges have capacity representing the $ that the person owes.
shouldn'tPay → sink edges have capacity representing the $ that person is owed.
shouldPay → shouldn'tPay edges have infinity capacity!

AN APPLICATION: BIPARTITE MATCHING

There are endless bipartite scenarios that could be translated into a Max-Flow problem!

A grou ... over a

C ... !

There are so many other types of problems!

Try coming up with other scenarios where Max-Flow & Ford-Fulkerson could be applied to solve the problem.

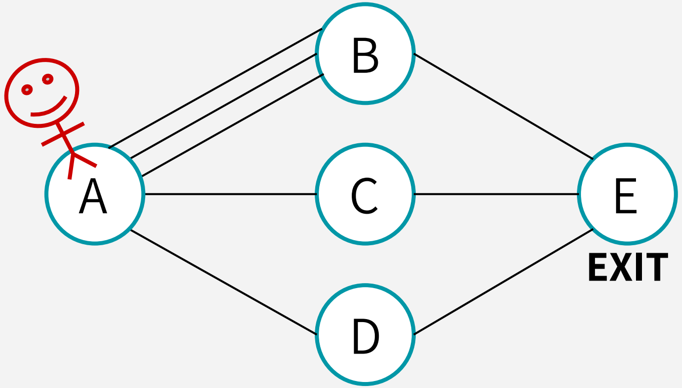
Create a source node s, and a sink node t.
Source ⟶ shouldPay edges have capacity representing the $ that the person owes.
shouldn't Pay ⟶ sink edges have capacity representing the $ that person is owed.
shouldPay ⟶ shouldn'tPay edges have infinity capacity!

سوال؟

یک کاربرد دیگر شاره پیشینه

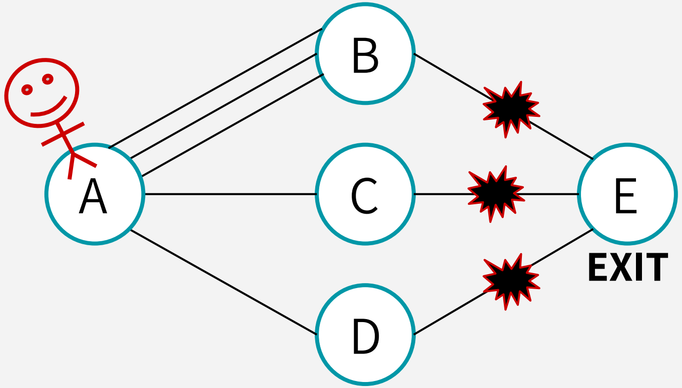
BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!



BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!

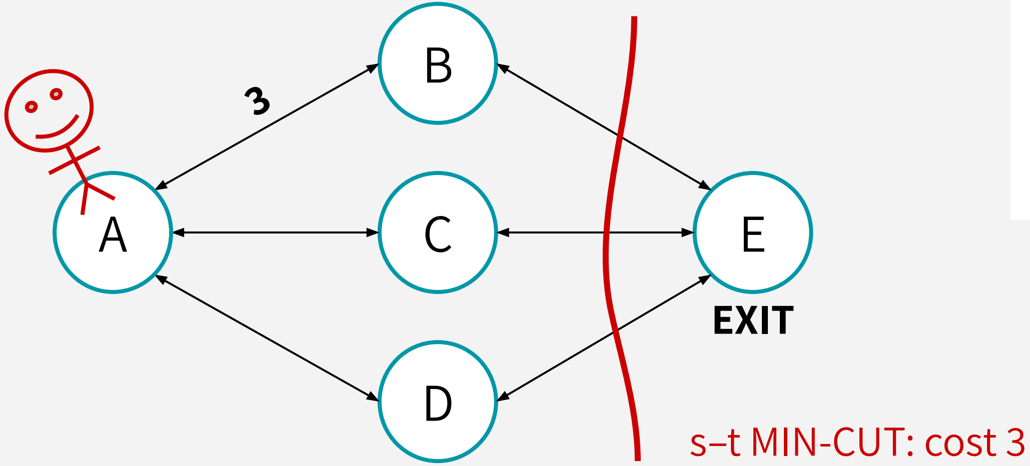


THINGS I NOTICE:

- undirected edges
- multi-edges
- cutting off resources (between a "source" and "sink")

BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!

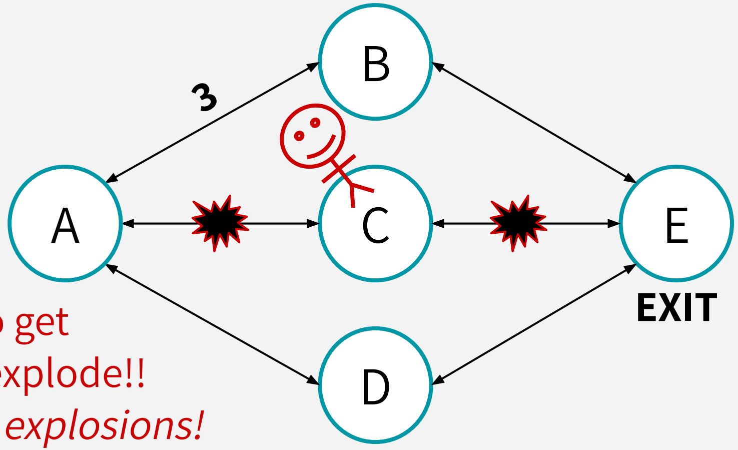


SOLVE AS s–t MIN-CUT!

- direct the edges
- multi-edges → weights
- s = thief current location
- t = exit

BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- ***BUT THE THIEF IS ON THE MOVE!!!***
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!



wait for thief to get
to C and then explode!!
→ *only costs 2 explosions!*

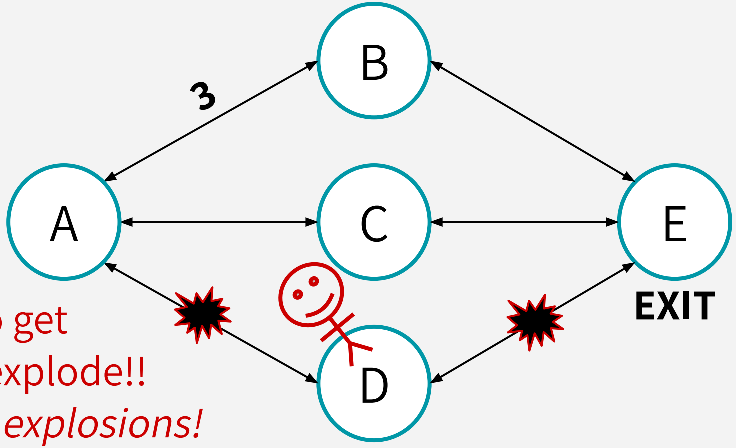
SOLVE AS s–t MIN-CUT??

- direct the edges
- multi-edges → weights
- s = thief current location
- t = exit

But now, s can change as the thief moves around, so we may want to delay explosions!

BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- ***BUT THE THIEF IS ON THE MOVE!!!***
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!



wait for thief to get
to D and then explode!!
→ *only costs 2 explosions!*

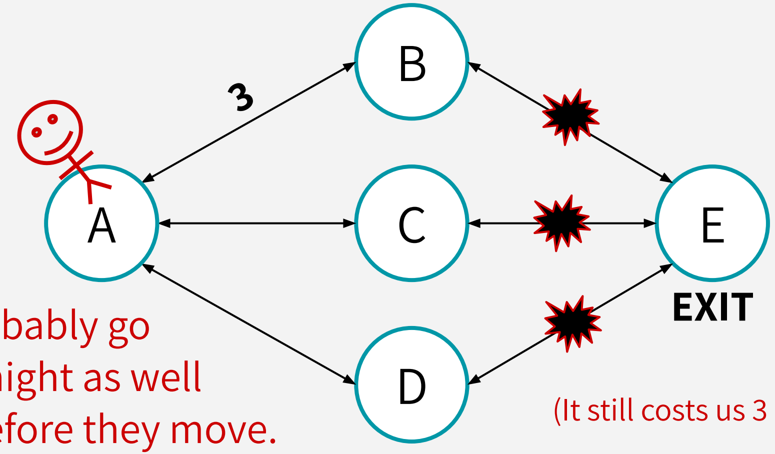
SOLVE AS s–t MIN-CUT??

- direct the edges
- multi-edges → weights
- s = thief current location
- t = exit

But now, s can change as the thief moves around, so we may want to delay explosions!

BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- ***BUT THE THIEF IS ON THE MOVE!!! (AND THE THIEF IS SMART)***
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!



Thief will probably go to B, so we might as well stop them before they move.

(It still costs us 3 explosions to stop the thief once they reach B)

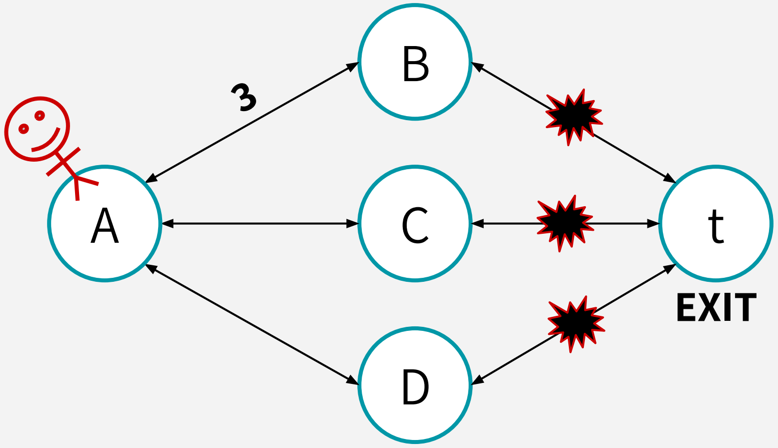
SOLVE AS s–t MIN-CUT??

- direct the edges
- multi-edges ⟶ weights
- s = thief current location
- t = exit

But now, s can change as the thief moves around, so we may want to delay explosions!

BONUS: STOPPING THIEVES

- Thief is inside an underground complex with rooms connected by tunnels
- ***BUT THE THIEF IS ON THE MOVE!!! (AND THE THIEF IS SMART)***
- There's 1 room that exits to the outside world where the thief can escape to
- We can track the thief's location, and we can stop the thief from escaping by closing tunnels (which requires mechanical effort)
- GOAL: close the minimum number of tunnels to trap the thief!



Thief starts at location x

- Find minCut(x,t)

For all paths p from x ⟶ t:

- minCutP = $\min_{v \in p}$ minCut(v,t)

maxMinCutP = largest of these minCutP

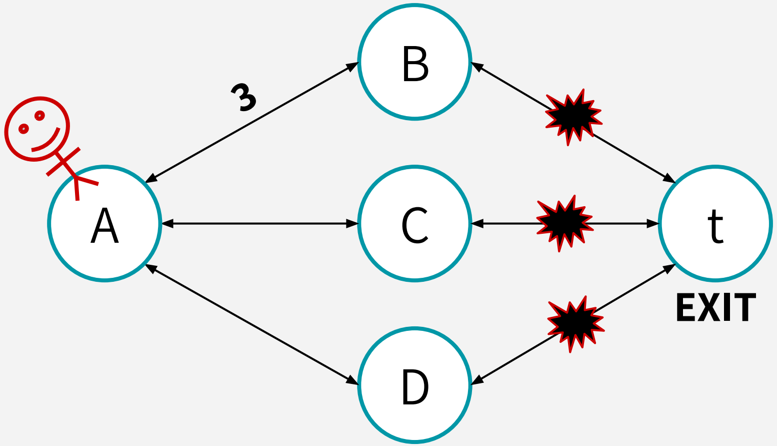
If minCut(x,t) >= maxMinCutP: delay explosions

- i.e. recurse on thief's next location

If minCut(x,t) < maxMinCutP: explode that min-cut!

BONUS: STOPPING THIEVES

- This was an example of cutting off resources (s–t min-cut application)
- This involved a “dynamic” source node, needed to remodel graph + edge weights
- Dynamic programming in nature!
  - Potentially would be recomputing minCut(v,t)many times → cache that!
- Smart & active thieves suck → min-max cleverness
- What if some tunnels can’t be closed?



Thief starts at location x
- Find minCut(x,t)

For all paths p from x → t:
- minCutP = $\min_{v \in p}$ minCut(v,t)

maxMinCutP = largest of these minCutP

If minCut(x,t) >= maxMinCutP: delay explosions
- i.e. recurse on thief’s next location

If minCut(x,t) < maxMinCutP: explode that min-cut!

سوال؟