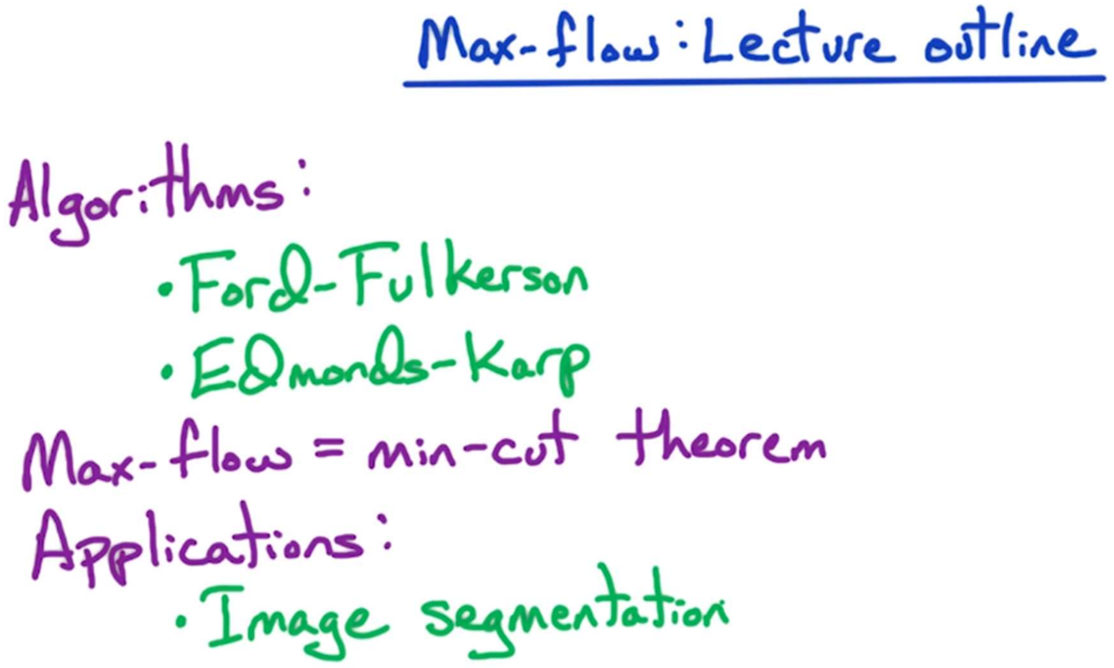
LESSON: MF1: Ford Fulkerson Algorithm
***
(Slide 1) Max-Flow
In this section, we'll look at the graph problem known as the Max-flow problem. It's a quite useful problem.

- We'll first look at the basic algorithm for the problem, known as the Ford-Fulkerson Algorithm, and then we'll prove correctness of the Ford-Fulkerson Algorithm.
- And along the way, we'll get the classic result known as the Max-flow-Min-cut theorem.
- Next, we'll look at the nice application of the Max-flow problem to computer vision known as the Image Segmentation problem.
- We'll also look at a more general version of the problem - Max-flow with additional demand constraints - and we'll see how to reduce this more general problem to the original Max-flow problem.
- Finally, we'll see a faster Algorithm for the Max-flow problem, known as the Edmonds-Karp Algorithm.

***

(Slide 2) Ford-Fulkerson Algorithm Lecture Outline



## Max-flow: Lecture outline

Algorithms:

- Ford-Fulkerson
- EDmonds-Karp

Max-flow = min-cut theorem

Applications:

- Image segmentation

This lesson is about the Max-flow problem.

- First, we'll focus on algorithms for solving the Max-flow problem. We'll start with the Ford-Fulkerson algorithm and then we'll look at the Edmonds-Karp algorithm. Both of these algorithms are very simple, but the analysis of the running time of Edmonds-Karp algorithm is a bit more involved.
- After that, we're going to look at the very beautiful and very important theorem known as the Max flow = Min-cut theorem. This theorem will imply correctness of the Ford-Fulkerson and Edmonds-Karp algorithms. It will also be key for several of our applications.
- In particular, we'll look at solving the Image Segmentation problem using the Max-flow problem. In the Image Segmentation problem, we're given an image and we want to separate it into the foreground and background components. We'll solve this using the Max-flow problem and utilizing the max flow equals min- cut theorem.

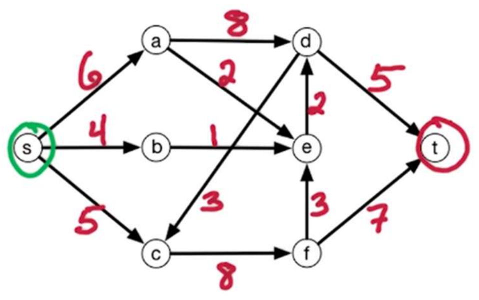
***

(Slide 3) Problem Setup

## Problem setup

Setting: sending supply from vertex s to t

Maximize amount sent without exceeding edge capacities



The general setting is that we're sending supply from vertex s to vertex t.

Now, what does this supply? The supply might be things like Internet traffic, or it might be a product. Now, the network is specified by a directed graph, and we have a designated start vertex s, and we have a designated end vertex t. And our goal is to maximize the amount of supply sent from s to t along the edges of this network, but we don't want to exceed the capacities of the edges.

So, for each edge we have a capacity of that edge. If you think of supply as an example such as Internet traffic then the capacity corresponds to the bandwidth of the link. If you think of a more physical example such as supply being oil, then we're maximizing the amount of oil sent from point s to point t in this network. And then the edges correspond to actual pipes, and then these numbers correspond to the capacity of these pipes.

***

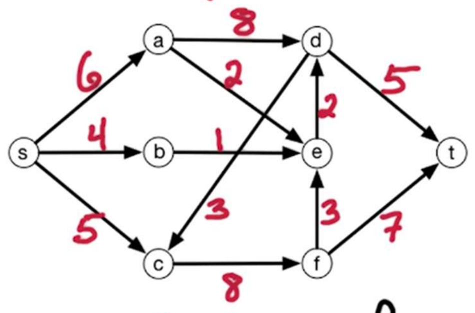
(Slide 4) Problem Formulation

## Problem formulation

**Flow network:**

directed graph $G=(V,E)$, designated $s,t \in V$

for each $e \in E$, capacity $c_e > 0$



Maximize flow s to t   $f_e$ = flow along e

Now let's formally state the Max-flow problem. The input to the Max-flow problem is what is called a **Flow network**. A flow network consists of a directed graph G.

Here's an example of a directed graph again. In this directed graph, we have two vertices labeled s and t, corresponding to the source and the sink for the flow, in our problem.

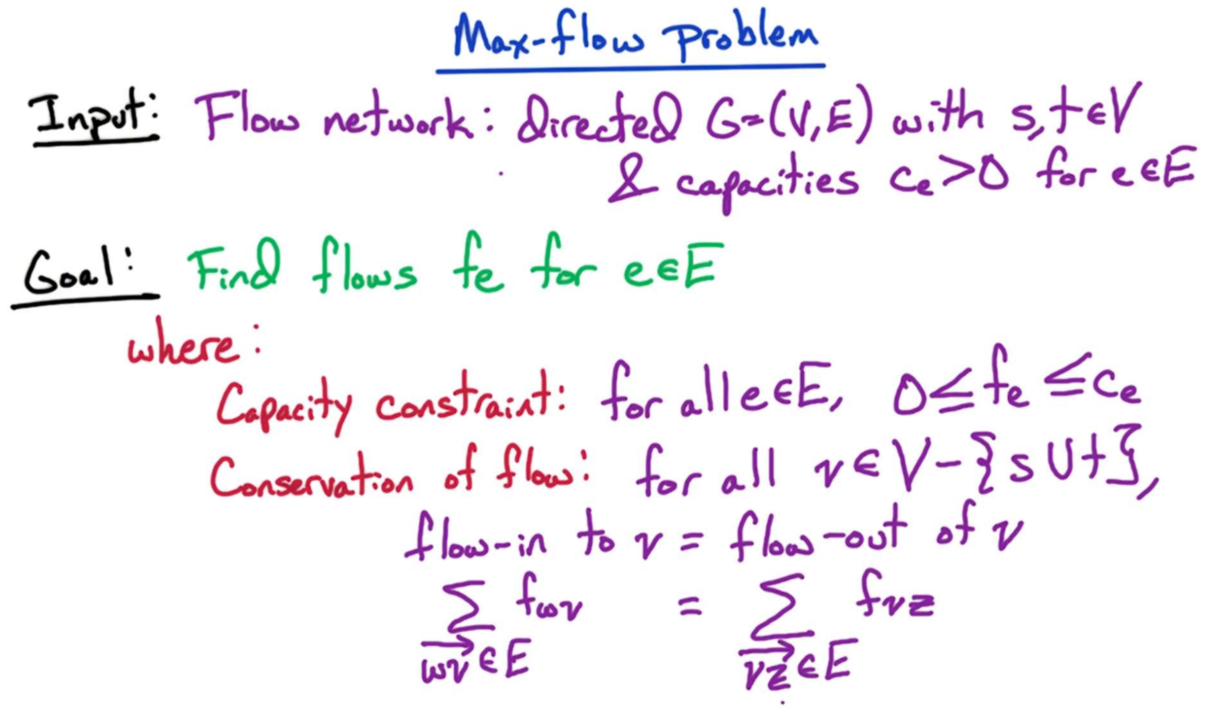
Finally, for each edge, we're given a capacity which is a positive number.

Here are the example capacities from before for this network.

And our goal is to maximize the flow which is sent from s to t. So, the flow originates at s and it ends at t. We use the variable $f_e$ to denote the flow along edge e. Now, these are the variables which we're trying to determine - our goal is to specify the flow along every edge so that we maximize the flow from s to t.

***

(Slide 5) Max-Flow Problem



## Max-flow Problem

**Input:** Flow network: directed $G=(V,E)$ with $s,t \in V$
& capacities $c_e > 0$ for $e \in E$

**Goal:** Find flows $f_e$ for $e \in E$

where:

Capacity constraint: for all $e \in E$, $0 \le f_e \le c_e$

Conservation of flow: for all $v \in V - \{s \cup t\}$,

flow-in to $v$ = flow-out of $v$

$$\sum_{\overrightarrow{wv} \in E} f_{wv} = \sum_{\overrightarrow{vz} \in E} f_{vz}$$

Here's the general formulation of the max-flow problem. So the input is a flow network. Flow network, once again, consists of a directed graph with a specified start vertex and end vertex, and for each edge, we're given a capacity which is a positive number specified by $c_e$.

Our goal is to find the maximum amount of flow that can be sent from s to t. This flow is specified by these variables $f_e$ for each edge e in the directed graph.

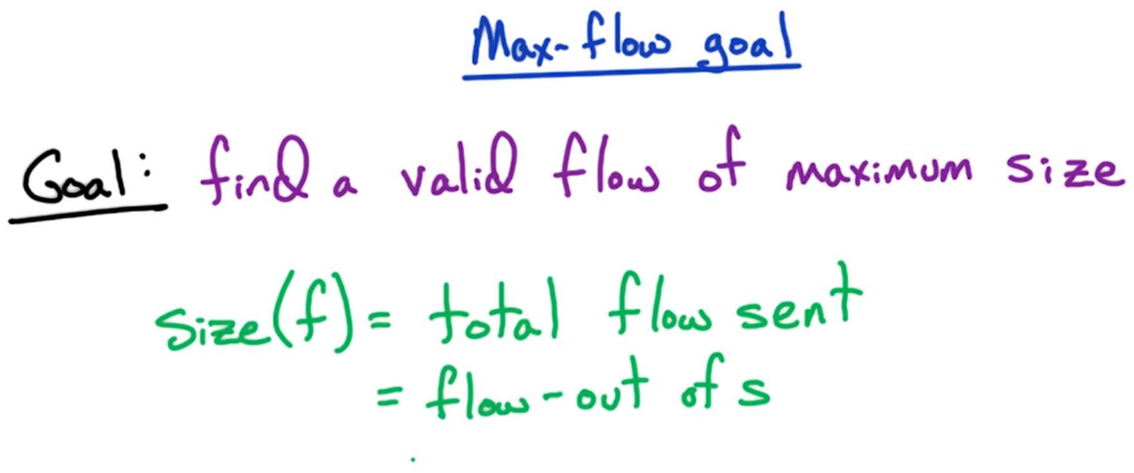
Now what are the constraints on these flows?

- The first constraint is the **capacity constraint**. For every edge, the flow along that edge cannot exceed the capacity of that edge, and the flow must be non-negative.
- The second constraint is **conservation of flow**. The flow originates at the source vertex s and ends at the sink vertex t. But, for all other vertices - so for all v in the vertex set minus s and t - the flow must be conserved: The flow into v must equal the flow out of v. Nothing can be lost or gained at vertex v. Formerly, the flow into v is the following: It's a sum over edges w -> v of the flow along this edge which is specified by $f_{wv}$, and the flow out of v is a sum over edges v -> z of the flow along that edge, which is specified by $f_{vz}$.

A valid flow is one which satisfies the capacity constraints and conserves the flow at every vertex except for the source and the sink.

***

(Slide 6) Max-Flow Goal



Max-flow goal

Goal: find a valid flow of maximum size

size(f) = total flow sent
= flow-out of s

Now our goal in the Max-flow problem is to find a valid flow. So one satisfying the capacity constraints and the conservation of flow constraints, and we want to find a valid flow of maximum size.

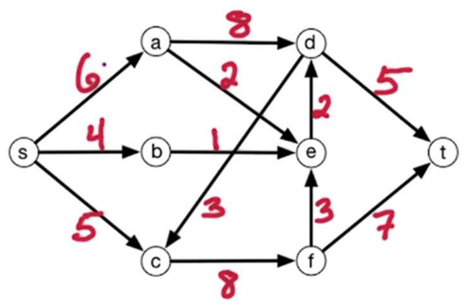
What exactly do we mean by maximum size? What is the size of a flow? Well, the size of the flow is the total flow sent. This can be measured by the total flow out of s or the flow into t.

***

(1st Slide 7) Quiz: Max-Flow Example

## Quiz: Max-flow example

Find a flow of maximum size satisfying constraints



To make sure you understand the formulation of the max-flow problem, let's take a look back at our earlier example.

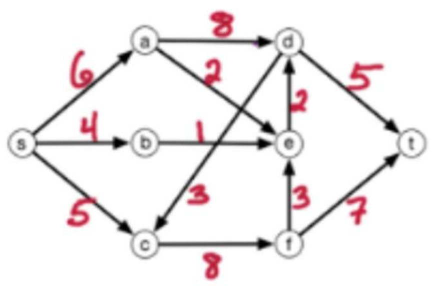
So here's a flow network with a specified start and end vertex, and we specified the capacity for every edge.

Now specify a flow. So, specify the flow along every edge where this flow is of maximum size - and the flow needs to be valid - it needs to satisfy the two constraints, which are the capacity constraints along every edge and the conservation of flow along every vertex except for vertices s and t.

MF1: Ford-Fulkerson Algorithm: Quiz: Max-Flow Example

## Quiz: Max-flow example

Find a flow of maximum size satisfying constraints



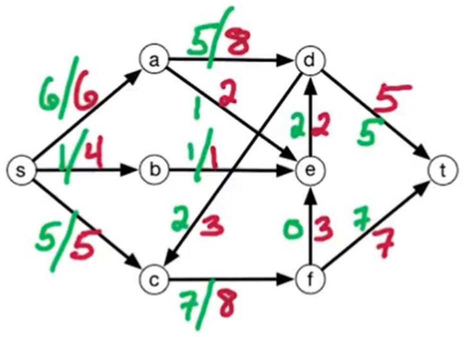
Question: Specify a valid integer flow value for each edge to maximize the total flow out of s and into t. Each flow value must be non-negative and must not exceed the capacity of the respective edge. Please enter a comma separated list for sa, sb, sc, ad, ae, be, cf, dc, dt, ed, fe, ft (in that order).

***

(2nd Slide 7) Quiz: Max-Flow Example (Answer)

## Solution: Max-flow example

Find a flow of maximum size satisfying constraints



size(f) = 12.

Now, here's a flow of maximum size. The flow is specified by the green numbers along every edge and their capacities, are specified by the red numbers along every edge.

Now, notice that, for every vertex, the flow in equals the flow out. For instance:

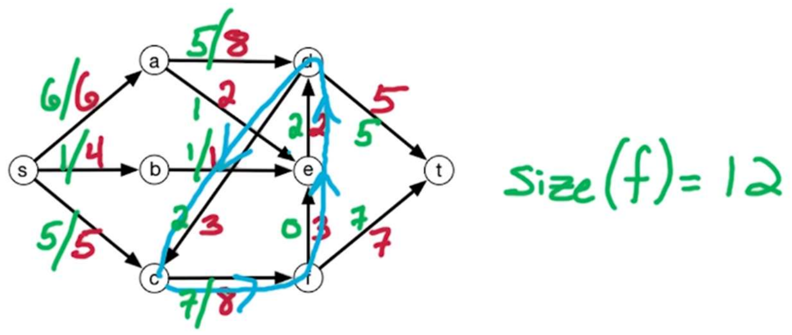
- For vertex c, the flow in is 7 and the flow out is 7.
- For vertex e, the flow in is 1 + 1 which is 2 and the flow out is 2.
- For vertex d, the flow in is 7 and the flow out is 7.
- And all of the flows along the edges are upper bounded by the capacities along those edges.
- And finally, the total flow size is of size 12. This is 12 units of flow coming out of s and there's 12 units of flow coming into t.

Now, for this example, why do we know that this is the maximum flow? – well, the total capacity of the edges coming into t is 12, so the maximum amount of flow that can come into t is at most 12 and we've achieved that; therefore, we've achieved the maximum flow.

***
(Slide 8) Cycles are OK

## Cycles are OK

Find a flow of maximum size satisfying constraints



One interesting feature of this example is that there is a cycle in this graph.

There's a cycle that goes from c -> f -> e -> d -> c.   This is a cycle in the original input graph in the flow network.

Now, we've seen for some other examples such the shortest path problem that cycles in the input graph can cause some issues.  In that formulation, it's negative weight cycles which cause problems. So a natural question is whether cycles are okay in a flow network for the Max-flow problem.

Notice that the flow network has a cycle but, our flow,  our maximum flow, is not utilizing this cycle.  And why is it not utilizing this cycle?  Well, suppose that I had one unit of flow coming into d and then I send one unit flow along this entire cycle back to t and then send that one unit out of d.  Well, I could just send it right from this edge into d and then out of  d directly without going around the cycle.   So, I could have bypassed the cycle.  So. there's no reason to utilize a cycle in a flow network.

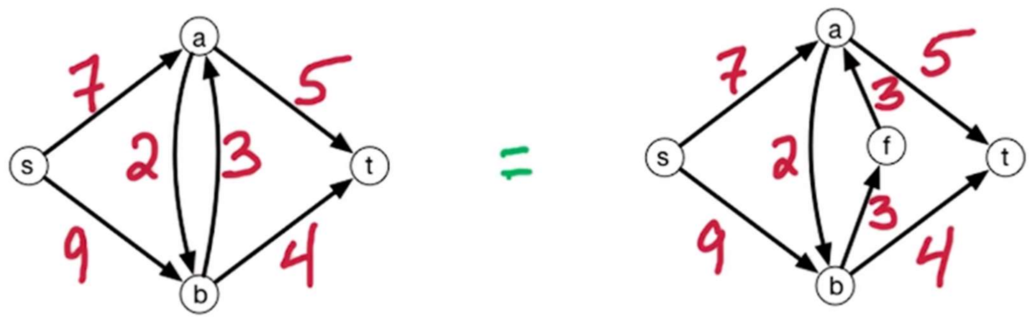
Now. I may want to utilize part of this cycle but I don't know a priori which part of the cycle I might want to utilize.

So, the main conclusion is that cycles are okay in a flow network.  In fact, they simplify the problem in some sense but the problem is  well-defined regardless of whether there are cycles in the flow network or not.

***

(Slide 9) Anti-parallel Edges

## Anti-parallel edges



Before we dive into algorithms for the Max-flow problem, let's look at one slight simplification of the input.

Here's an example flow network. Notice, that there are edges from a -> b, and also an edge from B -> A. These are called anti-parallel edges. Now, I'd like to remove these anti-parallel edges. Why we want to remove these anti-parallel edges will become clear later when we come up with our algorithms for the Max-flow problem. But for now, let's just say, we want to simplify the input a little bit.

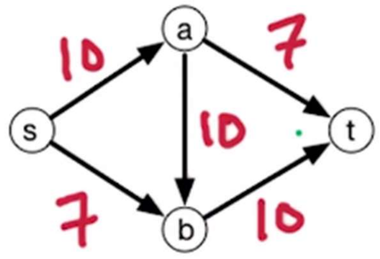
So I wanted to find an equivalent flow network, and work with that equivalent flow network. And, in that equivalent flow network, I don't have any anti-parallel edges.

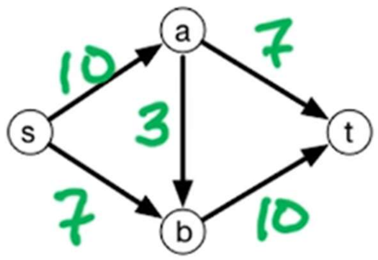
So let's put some capacities on these edges. So here are the edge capacities for this flow network. Now we wanted to find an equivalent flow network, and I wanted this flow network to not have any anti-parallel edges. I want to remove this anti-parallel edges here, and I want the max-flow and that equivalent network to be the same as the max-flow here.

This is the new network I want to consider. So I just split this edge from B to A, into a path of length two, and I added one auxiliary vertex, and the edge capacities stay the same. This single edge had a capacity of 3.  It gives both of these edges capacity 3, and notice that these two flow networks are equivalent to each other.  So, I can easily convert a solution to the max-flow problem on  one network to the other network and vice versa.

***
(Slide 10)

## Toy example





Now it'll be useful to have a small toy example to work with when we devise our algorithms.

So, let's look at this following example. It only has four vertices. It's quite easy to find a max flow in this network. Here's a max flow in this network - notice this flow is of size 17? Therefore, it's clearly maximum because it achieves the capacity into t and also the capacity out of s.

Now we're going to use this toy example as a running example.

***

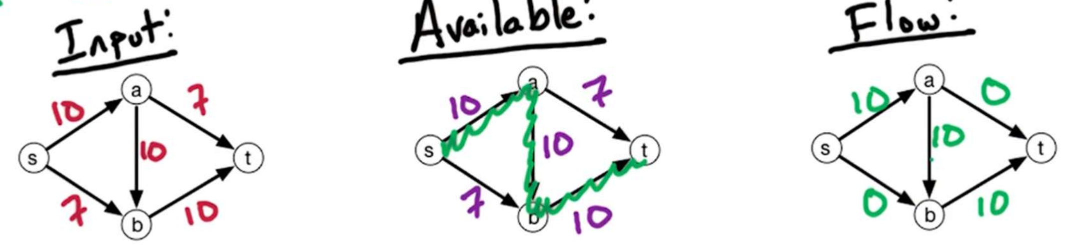
(Slide 11) Simple Algorithm

## Simple algorithm

Algorithm idea:

1. Start with $f_e = 0$ for all $e \in E$
2. Find st-path $P$ with available capacity
3. Let $c(P) = \min_{e \in P} (c_e - f_e)$
4. Augment $f$ by $c(P)$ along $P$

Repeat until No st-path



Let's take a look at a simple algorithm approach, and let's use the toy example from before as our running example.

Algorithm idea:

1. Let's start off by initializing the flow to 0 everywhere. So, we'll illustrate the current flow in this example. Then let's take a look at the available capacities along every edge. Since the flow is 0 everywhere, the available capacities will be the same as the capacities in the input flow network. So we'll use this middle copy of the graph to illustrate the available capacities along the edges.

2. Now in order to augment the flow in a valid manner, what we're going to do is we're going to try to find a path from s to t in this available capacity graph. So in this graph, we're going to find an s-t path - a path from s to t. Let's denote this path by P.

   How do we find such a path? Well, we can run BFS or DFS in this graph, it doesn't matter which one. We're not saying anything about which particular path we're looking for. We're just trying to finding find any path from s to t, where every edge has some available capacity. Let's consider this path from s to t in this graph. Now this path happens to be optimal in some sense. It has the one with the maximum available capacity, but that's not important for this example.

3. We're going to look at all edges in this path and we're going to look at the available capacity of that edge - we're going to look at the capacity along this edge, minus the current flow along this edge - it's the available capacity along this edge. We're going to minimize this. So this is going to be the maximum amount that we can send along this path.

4. So finally we're going to augment this flow, the current flow, by sending c(P) units of flow along this path P.

   Hence, this flow changes so that and then we now have 10 units of flow along this path, s -> a -> b -> t.

   So this is the new flow we're currently at and now we simply repeat.

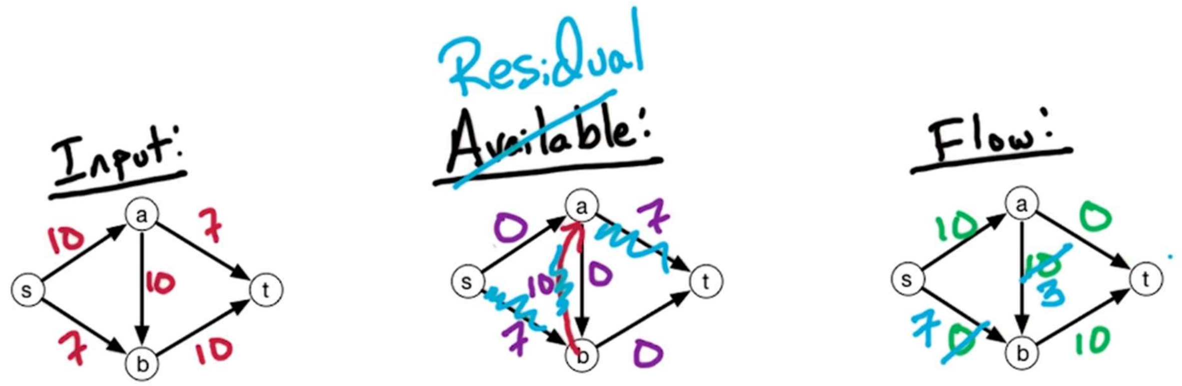
We redefine this graph of available capacities, we look for a path, s -> t, we augment along that path, we update the flow, we re-update this graph, and repeat. And we keep going until there's no s-t path in this graph.

So, let's update this graph for this current flow. Notice, that the available capacity for these three edges is now zero. So, in this graph there is no path from s to t anymore. So our algorithm is going to stop. There's no way to augment. So we're going to end with this flow, which is clearly sub-optimal. This is a flow of size 10, and we saw before there's a flow of size 17 in this graph.

***

(Slide 12) Backward Edges

## Backward edges



Now what are we missing in this graph of available capacities?  Well, if we reverse engineer a little bit, and we look at the optimal solution, we see that we want to augment along this path. So we'd like to augment the current flow along this path from s -> b -> a -> t.

In this way, we want to increase the flow along this edge, s -> b, and this edge,  a -> t, to seven. But what does it mean to augment along this edge from b -> a?  There's no such edge in this flow network.

We'll think about this as a pipe.  So when we send flow along this edge, a -> b, we're opening up faucet.  So we're sending more flow along this pipe.  If we send flow from b -> a, it's like we're closing the faucet,  so we're sending less flow from a -> b.

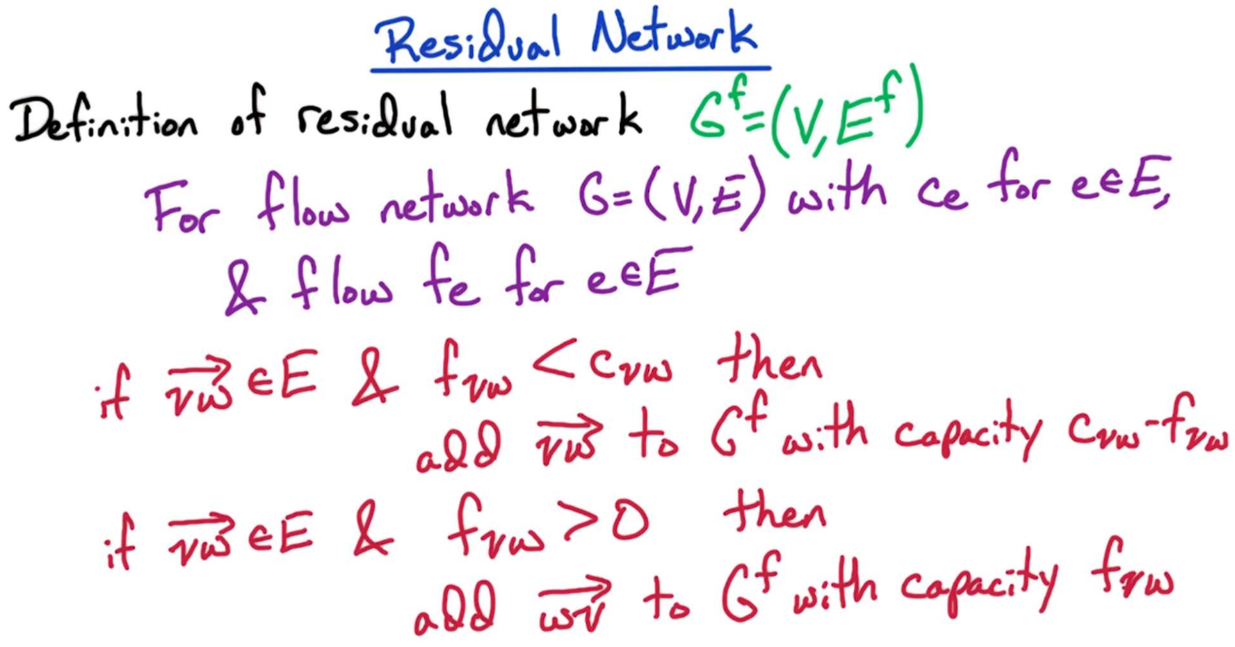
The point is that this graph of available capacities is not properly  representing everything that's possible to do from this current flow.  In particular, we want to have an edge from b -> a,  and this edge should have available capacity 10.  Why 10?  Because the current flow from a -> b is 10 units.  So, if we send at most 10 units of flow from b -> a,  that corresponds to decreasing the flow from a -> b.

Now if we look at this graph,  this is no longer the graph of available capacities.  This graph is called the residual network.  And, in this graph, we can find a from path s -> t.  Namely, we can find this path that we're looking for, you go from s -> b -> a -> t. The capacity of this path is seven units.  So we can augment the flow by seven units along this path.  So the flow from s -> b will change to seven units,  we'll send seven units from b -> a by decreasing the flow from a -> b by seven units,  so it will go down from 10 to 4.  And from a -> t,  the flow will increase by 7 units.  And then, we'll be at a Max-flow for this example.

So the key idea is to look at the residual network instead of the network of available capacities.

***

(Slide 13) Residual Network



## Residual Network

Definition of residual network $G^f=(V,E^f)$

For flow network $G=(V,E)$ with $c_e$ for $e\in E$,

& flow $f_e$ for $e\in E$

if $\overrightarrow{vw}\in E$ & $f_{vw}<c_{vw}$ then

add $\overrightarrow{vw}$ to $G^f$ with capacity $c_{vw}-f_{vw}$

if $\overrightarrow{vw}\in E$ & $f_{vw}>0$ then

add $\overrightarrow{wv}$ to $G^f$ with capacity $f_{vw}$

Let's formally define a residual network.

First off, we have our input to the max-flow problem which is the flow network, the directed graph with capacities for each edge, and our residual network will be a function of the current flow.

So $f_e$ for $e \in E$ is the current flow. So we'll denote the residual network as this graph $G^f$ (G superscript f) because it is a function of the current flow f. This will be a directed graph. The vertices will be the same as the flow network, but the edges will differ.

So the edges will be a function of the current flow. These edges will also have weights.
And, even the actual edges that exist will be dependent on the current flow - and these might differ from the flow network or edges.

For example, in the previous toy example, we saw that the flow network had an edge from a -> b, but the residual network will have an edge from b -> a.

So how, specifically, do we define the edges of this residual network? Well, if there's an edge from v -> w in the original flow network, and this edge has some available capacity – (ex: so, the flow along this edge is strictly smaller than its available capacity) - then we add the same edge to the residual network where the capacity in this residual network is the **available** capacity. So it's the total capacity minus the current flow.

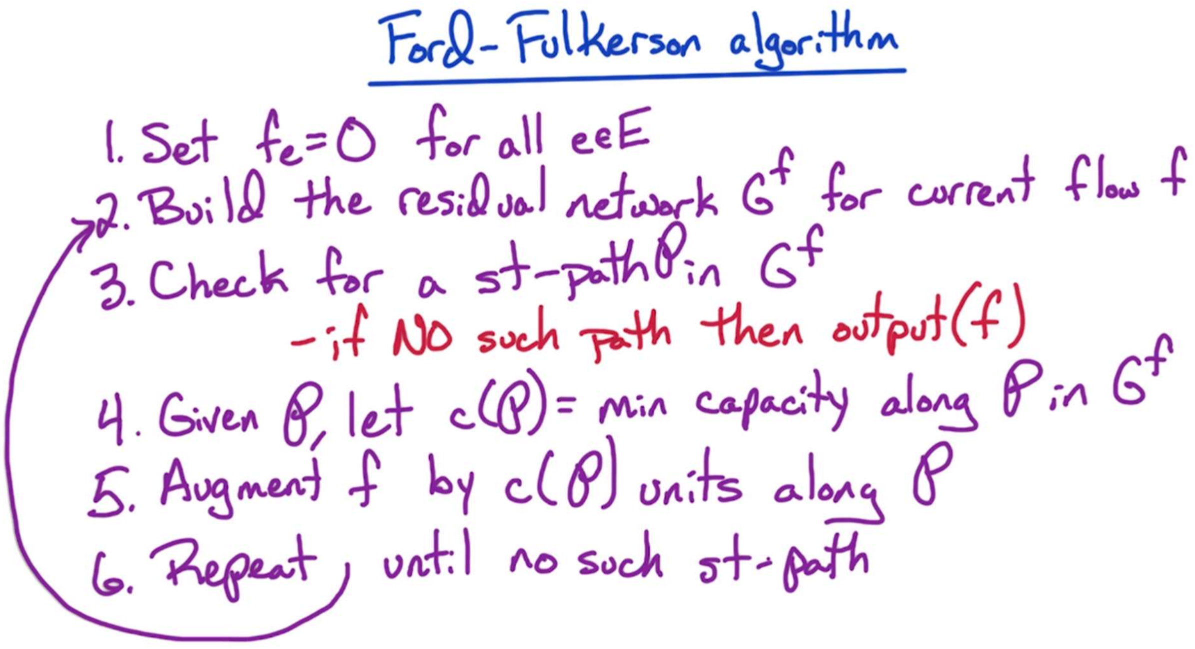
These are, in some sense, the forward edges. These are the original type of edges we considered when we talked about available capacity network. Now, for this edge in the original flow network, if the flow is positive so there is some flow going along this edge, then we add the reverse of the edge to the residual network. Notice that this was the edge v -> w and now we added the edge from w -> v.

This is the backward edge that we talked about earlier in the toy example. And the capacity of this backward edge is the current flow along the edge v -> w so we can send most $f_{vw}$ units of flow along this edge from w -> v, and we send this flow from w -> v by decreasing the flow along v -> w.

Now, since we sometimes have the forward edge and the backward edge, this is why, in our original flow network, we removed antiparallel edges. This way we're allowed to add the forward edge and the backward edge and we don't have to worry about any inequalities.

***

(Slide 14) Ford-Fulkerson Algorithm



Ford-Fulkerson algorithm

1. Set $f_e = 0$ for all $e \in E$
2. Build the residual network $G^f$ for current flow $f$
3. Check for a st-path $P$ in $G^f$
   - if NO such path then output($f$)
4. Given $P$, let $c(P)$ = min capacity along $P$ in $G^f$
5. Augment $f$ by $c(P)$ units along $P$
6. Repeat until no such st-path

Now we can state the Ford-Fulkerson algorithm. This is going to follow the same basic approach we did before, but instead of considering the graph of available capacities, we're instead going to consider the residual network.

Ford-Fulkerson algorithm:

1. We're going to start by initializing the flow to zero everywhere

2. Then we build the residual network for the current flow.

   Initially, the residual network will be the same as the input flow network.

3. Now we look for a path from s -> t in this residual network.

   How do we look for the path? Once again, we use DFS or BFS. If there is no such path, then we stop the algorithm and we output the current flow as our output as the max flow for this graph.

4. If instead we found such a path - let's first denote this path by P, and let's define the capacity of this path - let C(P) be the minimum capacity along this path in the residual network.

So, look at the edges in this path - look at the capacities of these edges in the residual network.

5. Finally, we augment the current flow by c(P) units along this path.

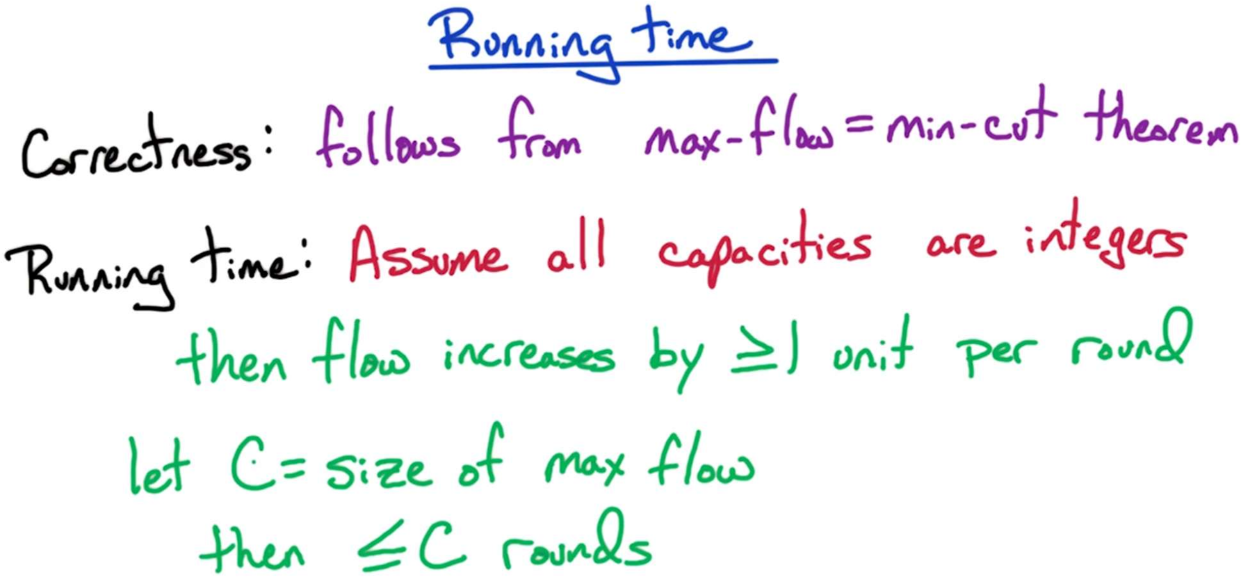
   What exactly does it mean to augment the flow? Well, for every forward edge we increase the flow along that edge by this amount c(P). For every backward edge, we decrease the flow in the other direction, in the reverse direction by this amount c(P).

6. Finally, we'll repeat from step 2, build the residual network and check for an st-path and we continue repeating until there is no such st-path.

   And then we output the current flow as our solution to the Max-flow problem (see Step 3).

***

(Slide 15) Running Time



## Running time

Correctness: follows from max-flow = min-cut theorem

Running time: Assume all capacities are integers

then flow increases by $\geq 1$ unit per round

let $C$ = size of max flow

then $\leq C$ rounds

Now the proof of correctness of this algorithm - the fact that it does output flow which is equal to the Max-flow - this will follow from the Max-flow=Min-cut theorem which we'll see later. So we're going to skip the proof of correctness for now. Let's instead focus on the running time with this algorithm.

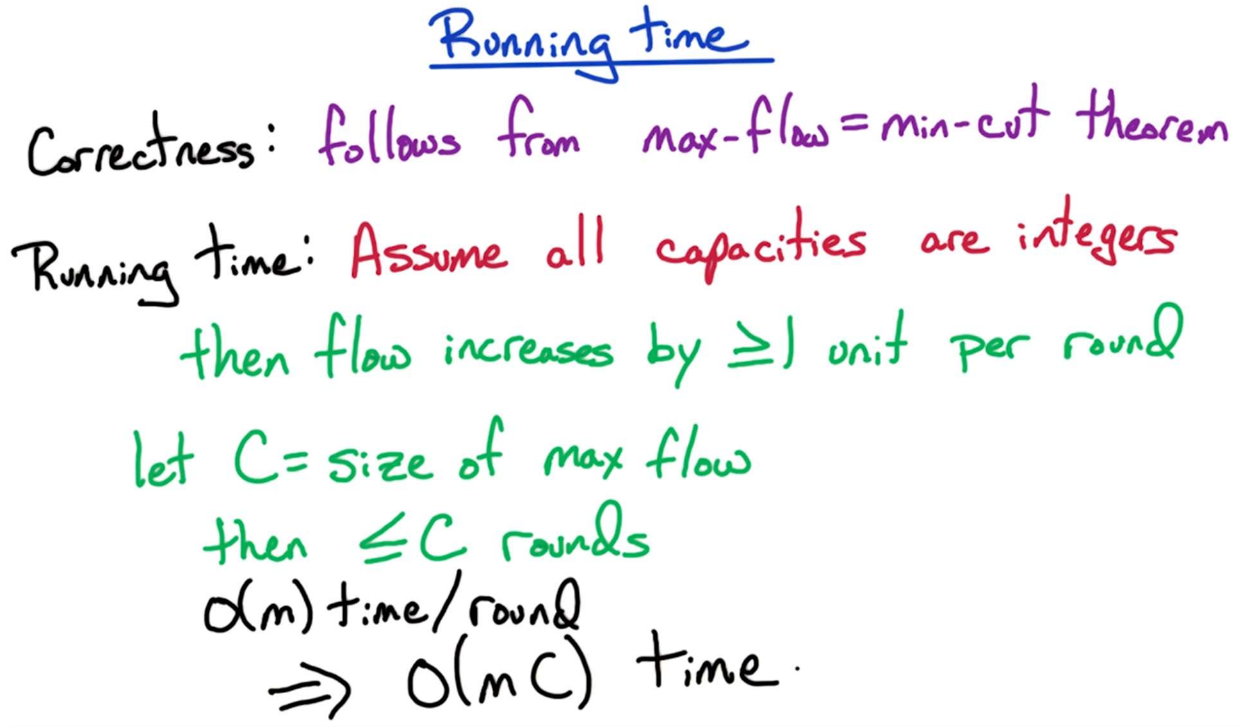
In order to analyze the running time, we need to make a huge assumption. We need to assume that all the capacities are integers. Only under this assumption can we make a claim about the running time with a Ford-Fulkerson algorithm. Later, we'll see more sophisticated algorithms namely, Edmonds-Karp algorithm, which eliminates this assumption.

The main point of this assumption is that when we augment the flow, we're going to augment it by an integer amount. So therefore, the capacities in the residual network will maintain as integers, and then every round will continue to augment by an integer amount. Now, if all the capacities in the residual network are integers, therefore they are all at least one, since they're all positive. Then, the flow is going to augment by at least one unit, because the capacity of this path will be at least one unit. So the flow is going to increase by at least one unit for every round of the Ford-Fulkerson algorithm.

Now how many rounds do we have? Let's let C denote the size of the Max-flow. This is what we're trying to find. Well then, we have at most C rounds of the algorithm, because in every round, the flow increases by at least one unit. The maximum flow that we're trying to find is of size C, so we're going to get at most C rounds.

***

(Slide 16) Time per Round



Running time

Correctness: follows from max-flow = min-cut theorem

Running time: Assume all capacities are integers

then flow increases by $\geq 1$ unit per round

let $C$ = size of max flow

then $\leq C$ rounds

$O(m)$ time/round

$\Rightarrow O(mC)$ time.

Now, here again is Ford-Fulkerson algorithm, and now the question is, what is the time needed for one round of the algorithm?

Now, one round of the algorithm includes steps 2 through 5 above (see Slide 14). In particular, we first build the residual network, then we check for a path, and then we augment along that path.

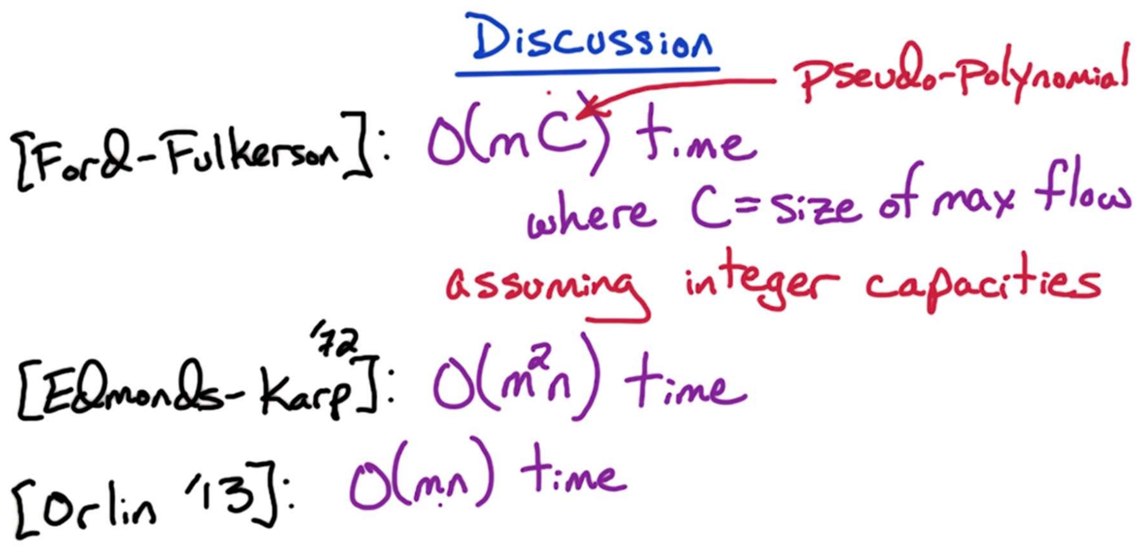
Now, originally, the residual network is just the input network. Now, given the residual network from the previous round, how long does it take us to update the residual network, given this augmentation along this path P? – well, the residual network is simply going to change along this path P. It might change along the forward edges or the backward edges, or both.

This path P is of length at most n-1 edges, so it's going to take us O(n) time to update the residual network (steps 2-5). How do we check for s-t path in this residual network? Well, we run either DFS or BFS. The running time is linear, O(n+m). n is the number of vertices, m is the number of edges. Lets assume the number of edges is at least n-1, then this is bounded by O(m) and also augmenting the flow along this path also takes O(m) time. So, one round of the algorithm is dominated by this checking for an st-path which takes O(m) time.

Now, going back to our running time analysis what we  concluded was that it takes O(m) time per round of the algorithm.  As discussed before, there are at most capital C rounds of the algorithm.  Since in every round the flow increases by at least one unit, therefore, we can conclude that Ford-Fulkerson algorithm requires time O(mC).

***

(Slide 17) Discussion



## Discussion

pseudo-polynomial

[Ford-Fulkerson]: $O(mC)$ time
where $C$ = size of max flow
assuming integer capacities

[Edmonds-Karp '72]: $O(m^2 n)$ time

[Orlin '13]: $O(mn)$ time

What we just saw is a Ford-Fulkerson algorithm, takes running time, O(mC), where C is a size of the max-flow. But this assume that the capacities were integer values.

Now there are two problems or unpleasant aspects of this. The first is that, we're assuming that the capacities are integer values. The other is that, the running time depends on the output. It depends on the size of the max-flow, which depends on the size of the capacities in the input.

Since the running time of this algorithm depends on the numbers, the capacities in the input, we say that this running time is pseudo-polynomial. This is much like the situation for Knapsack. And there we discussed that the running time should depend on the log of these numbers, and ideally, the running time should be independent of these numbers.

What we're going to see next is the Edmonds-Karp algorithm. This is very similar to the Ford-Fulkerson algorithm. It's got the same paradigm, but in the Ford-Fulkerson algorithm, we just find any path from s -> t in the residual network, and we can run DFS or BFS to find such a path. In the Edmonds-Karp algorithm, we take the shortest path from s -> t. Shortest means minimum number of edges. We don't care about the weights on the edges.

To find such a path, we just run BFS, and then, what can prove is that the number of rounds is going to be at most m x n, since each round, again, takes O(m) time, the total running time will be $O(m^2 n)$. So the running time will be independent of the size of the max-flow, and we no longer need this assumption that the capacities are integer values. Finally, we'll point out that Orlin has a current past algorithm from 2013,  in contrast, the Edmonds-Karp algorithm is from 1972.  So roughly 40 years later.  And he achieves a running time of order m x n. This is currently the best for general graphs for the exact solution of the max-flow problem.