Network flow

References

Jon Kleinberg, Eva Tardos «Algorithm design» ,

Chapter 7

The Maximum-Flow Problem and the Ford-Fulkerson Algorithm

The Problem

One often uses graphs to model transportation networks—networks whose edges carry some sort of traffic and whose nodes act as “switches” passing traffic between different edges.

Consider, for example,

a highway system in which the edges are highways and the nodes are interchanges;

a computer network in which the edges are links that can carry packets and the nodes are switches;

a fluid network in which edges are pipes that carry liquid, and the nodes are junctures where pipes are plugged together.

Network models of this type have several ingredients:

- capacities on the edges, indicating how much traffic they can carry;
- source nodes in the graph, which generate traffic;
- sink (or destination) nodes in the graph, which can “absorb” traffic as it arrives;
- the traffic itself, which is transmitted across the edges.

Flow Networks

We'll be considering graphs of this form, and we refer to the traffic as flow—an abstract entity that

is generated at source nodes,

transmitted across edges,

and absorbed at sink nodes.

Formally, we'll say that a flow network is a directed graph $G = (V, E)$ with the following features.

Associated with each edge e is a capacity, which is a nonnegative number that we denote c_e.

There is a single source node $s \in V$.

There is a single sink node $t \in V$.

Nodes other than s and t will be called internal nodes.

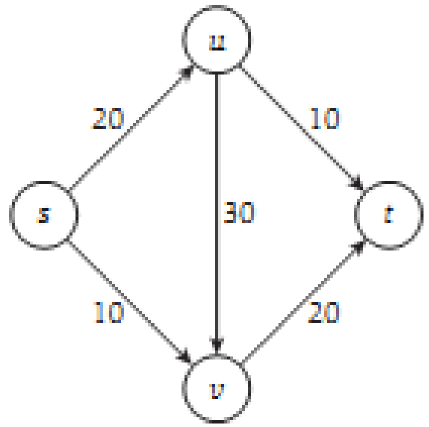


Fig.1 A flow network, with source s and sink t. The numbers next to the edges are the capacities.

We will make 2 assumptions about the flow networks we deal with:

1) no edge enters the source *s* and no edge leaves the sink *t*;

2) there is at least 1 edge incident to each node;

3) all capacities are integers.

These assumptions make things cleaner to think about, and while they eliminate a few pathologies, they preserve essentially all the issues we want to think about.

Defining Flow

An *s-t* flow is a function *f* that maps each edge *e* to a nonnegative real number $f : E \rightarrow \mathbb{R}^+$;
the value *f*(*e*) intuitively represents the amount of flow carried by edge *e*.

A flow *f* must satisfy the following 2 properties.

(i) (Capacity conditions) For each $e \in E$, we have $0 \leq f(e) \leq c_e$.

(ii) (Conservation conditions) For each node *v* other than *s* and *t*, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

$\sum_{e \text{ into } v} f(e)$ sums the flow value *f*(*e*) over all edges entering node *v*,

$\sum_{e \text{ out of } v} f(e)$ is the sum of flow values over all edges leaving node *v*.

Thus the flow on an edge cannot exceed the capacity of the edge.

For every node other than the source and the sink, the amount of flow entering must equal the amount of flow leaving.

The source has no entering edges (by our assumption), but it is allowed to have flow going out; in other words, it can generate flow.

Symmetrically, the sink is allowed to have flow coming in, even though it has no edges leaving it.

The value of a flow *f* , denoted *v*(*f*), is defined to be the amount of flow generated at the source:

$$v(f) = \sum_{e \text{ out of } s} f(e).$$

To make the notation more compact, we define

$$f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$$

$$f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e).$$

We can extend this to sets of vertices;

if $S \subseteq V$, we define

$$f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e)$$

$$f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e).$$

In this terminology, the conservation condition for nodes $v \neq s, t$

becomes $f^{in}(v) = f^{out}(v)$;

and we can write $v(f) = f^{\text{out}}(s)$.

The Maximum-Flow Problem

The basic algorithmic problem we will consider is the following:

Given a flow network, find a flow of maximum possible value.

As we think about designing algorithms for this problem, it's useful to consider how the structure of the flow network places upper bounds on the maximum value of an *s*-*t* flow.

Here is a basic "obstacle" to the existence of large flows:

Suppose we divide the nodes of the graph into 2 sets, *A* and *B*, so that $s \in A$ and $t \in B$.

Then, intuitively, any flow that goes from *s* to *t* must cross from *A* into *B* at some point, and thereby use up some of the edge capacity from *A* to *B*.

This suggests that each such "cut" of the graph puts a bound on the maximum possible flow value.

The maximum-flow algorithm that we develop here will be intertwined with a proof that the maximum-flow value equals the minimum capacity of any such division, called the minimum cut.

As a bonus, our algorithm will also compute the minimum cut.

We will see that the problem of finding cuts of minimum capacity in a flow network turns out to be as valuable, from the point of view of applications, as that of finding a maximum flow.

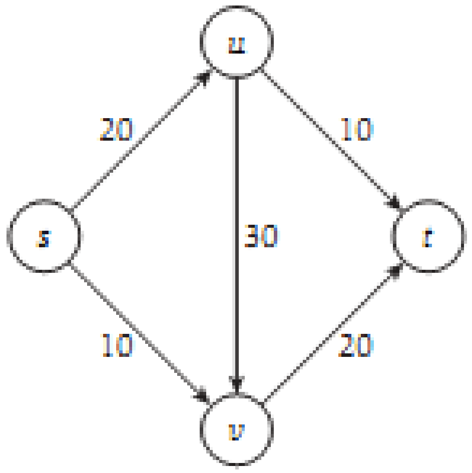
Designing the Algorithm

Suppose we wanted to find a maximum flow in a network.

How should we go about doing this?

Suppose we start with zero flow: $f(e) = 0$ for all e.

Clearly this respects the capacity and conservation conditions; the problem is that its value is 0.



(a)

We now try to increase the value of *f* by "pushing" flow along a path from *s* to *t*, up to the limits imposed by the edge capacities.

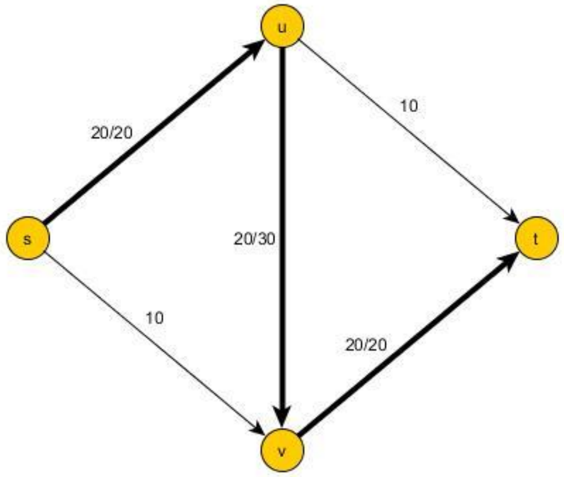
Thus, we might choose the path consisting of the edges {(*s*, *u*), (*u*, *v*), (*v*, *t*)} and increase the flow on each of these edges to 20, and leave *f*(*e*) = 0 for the other two.

In this way, we still respect the capacity conditions—since we only set the flow as high as the edge capacities would allow

—and the conservation conditions—since when we increase flow on an edge entering an internal node, we also increase it on an edge leaving the node.

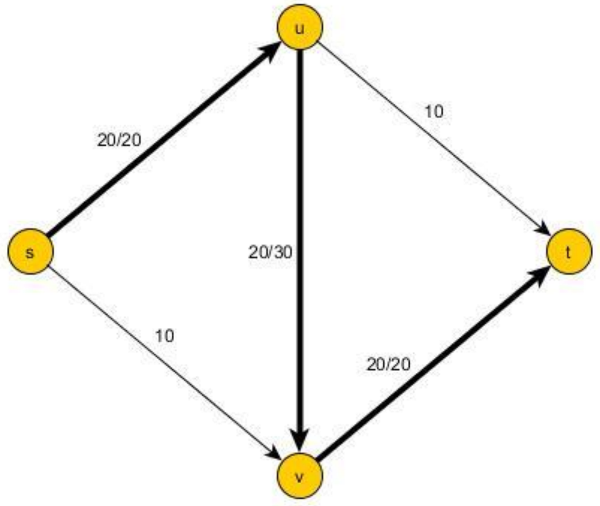
Now, the value of our flow is 20, and we can ask:

Is this the maximum possible for the given graph?



If we think about it, we see that the answer is no, since it is possible to construct a flow of value 30.

The problem is that we're now stuck—there is no *s*-*t* path on which we can directly push flow without exceeding some capacity—and yet we do not have a maximum flow.



What we need is a more general way of pushing flow from *s* to *t*, so that in a situation such as this, we have a way to increase the value of the current flow.

Essentially, we'd like to perform the following operation denoted by a dotted line.

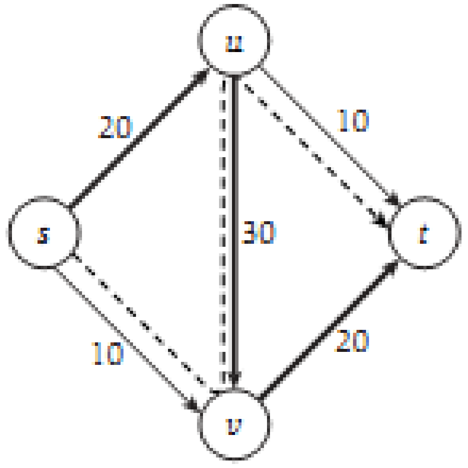
We push 10 units of flow along (*s*, *v*); this now results in too much flow coming into *v (20 + 10)*.

So we "undo" 10 units of flow on *(u*, *v*); this restores the conservation condition at *v* but results in too little flow leaving *u*.

So, finally, we push 10 units of flow along (*u*, *t*), restoring the conservation condition at u.

We now have a valid flow, and its value is 30.

The dark edges are carrying flow before the operation, and the dashed edges form the new kind of augmentation.

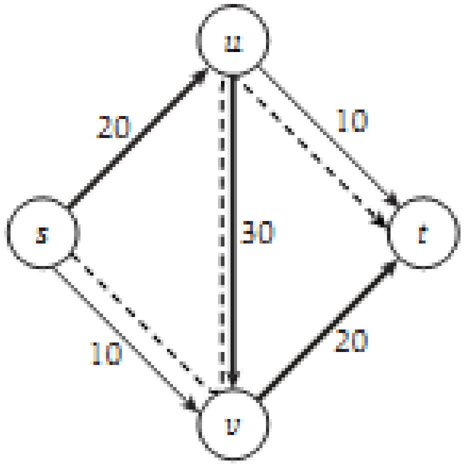


(c)

This is a more general way of pushing flow:

We can push forward on edges with leftover capacity, and we can push backward on edges that are already carrying flow, to divert it in a different direction.

We now define the residual graph, which provides a systematic way to search for forward-backward operations such as this.



(c)

The Residual Graph

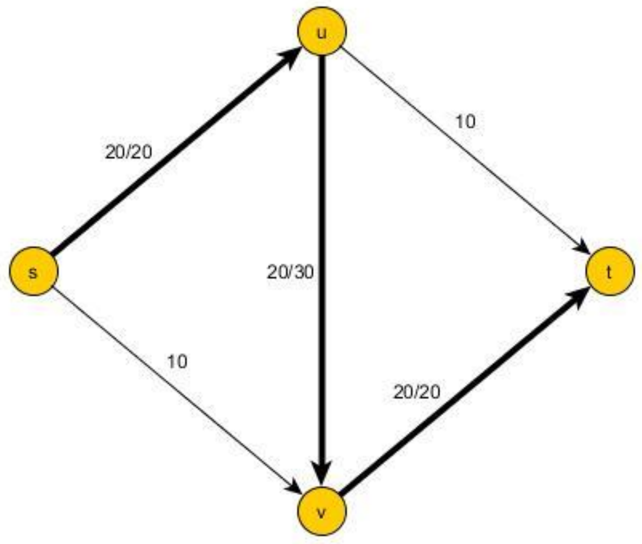
Given a flow network G, and a flow f on G, we define the residual graph G_f (остаточный граф, остаточная сеть) of G with respect to f as follows.

1) The node set of G_f is the same as that of G.

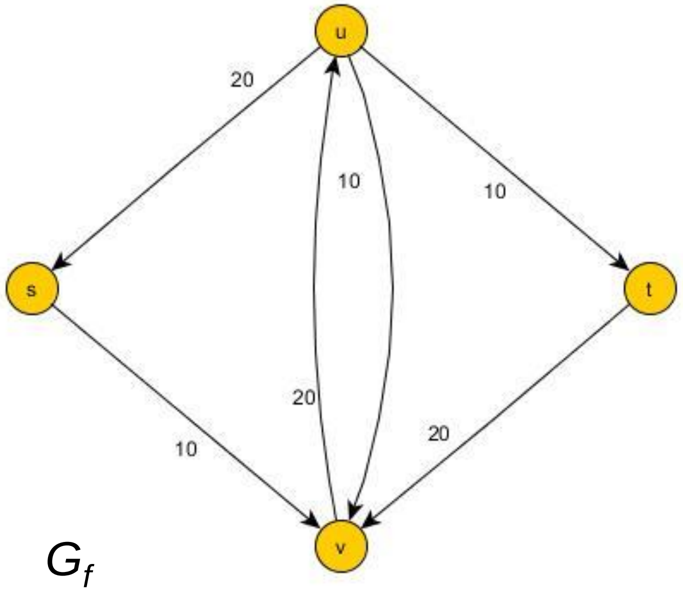
2) For each edge $e = (u, v)$ of G on which $f(e) < c_e$, there are $c_e - f(e)$ "leftover" units of capacity on which we could try pushing flow forward.

So we include the edge $e = (u, v)$ in G_f , with a capacity of $c_e - f(e)$.

We will call edges included this way forward edges (прямые ребра).



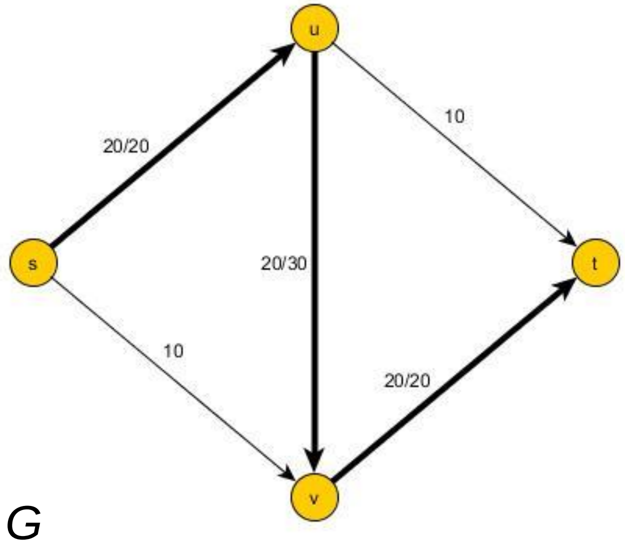
G

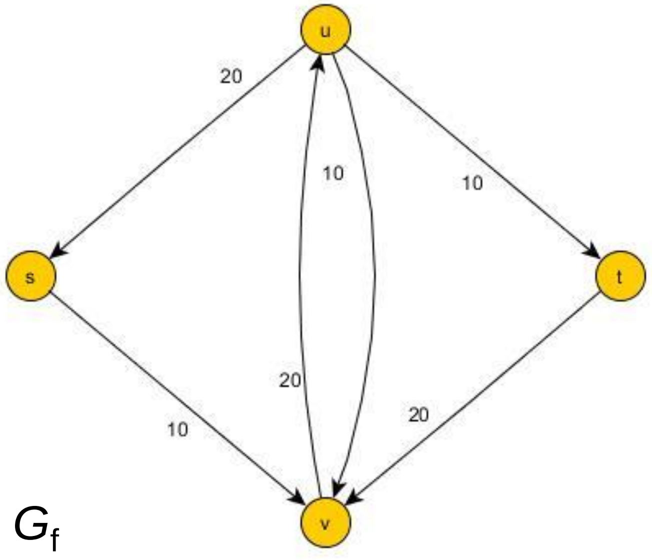


G_f

The Residual Graph

3) For each edge $e = (u, v)$ of G on which $f(e) > 0$, there are $f(e)$ units of flow that we can "undo" if we want to, by pushing flow backward.
So we include the edge $e = (v, u)$ in G_f , with a capacity of $f(e)$.
Note that e has the same ends as e, but its direction is reversed; we will call edges included this way backward edges (обратные ребра).





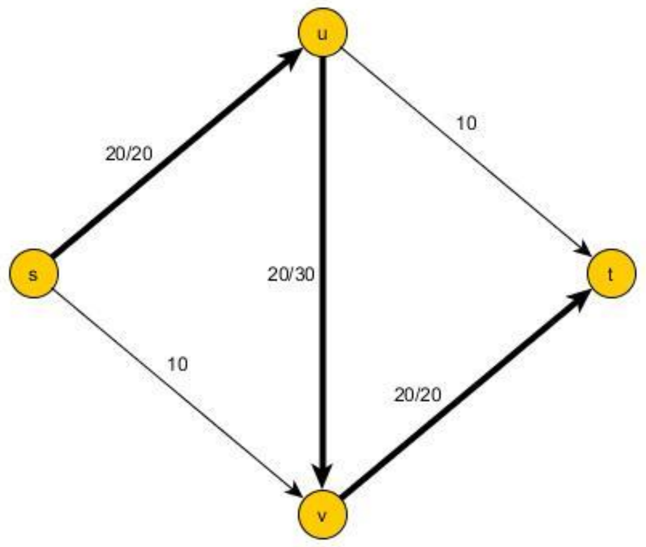
This completes the definition of the residual graph G_f .

Note that each edge *e* in *G* can give rise to 1 or 2 edges in G_f :

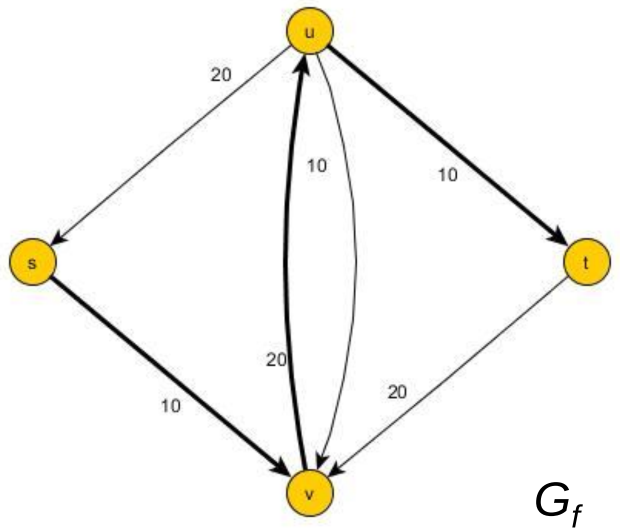
If $0 < f(e) < c_e$ it results in both a *forward edge and a backward edge* being included in G_f .

Thus G_f has at most twice as many edges as *G*: $|E_f| \leq 2|E|$.

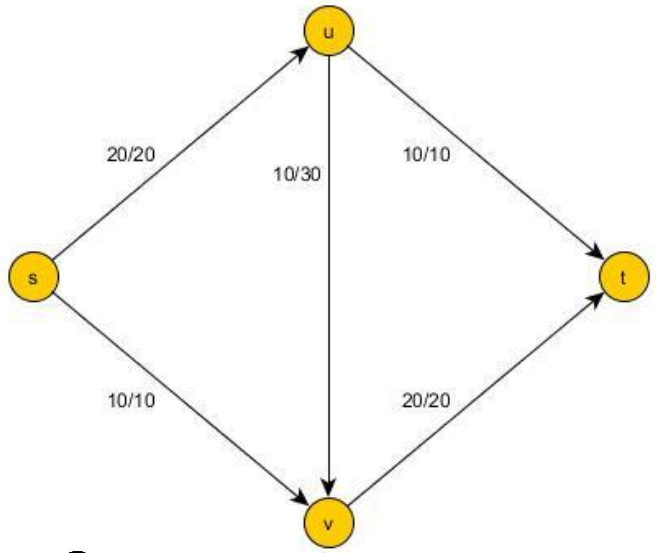
We will sometimes refer to the capacity of an edge in the residual graph as a *residual capacity,* to help distinguish it from the capacity of the corresponding edge in the original flow network *G*.


u
s
t
v
20/20
10
20/30
10
20/20

G


u
s
t
v
20
10
10
20
10
20

G_f


u
s
t
v
20/20
10/30
10/10
10/10
20/20

G

Augmenting Paths in a Residual Graph

Now we want to make precise the way in which we push flow from s to t in G_f.

Let P be a simple s-t path in G_f—that is, P does not visit any node more than once.

We define *bottleneck(P, f)* to be the minimum residual capacity of any edge on P, with respect to the flow f .

We now define the following operation *augment(f, P)*, which yields a new flow f in G.

```
augment(f, P)
Let b = bottleneck(P, f )
For each edge (u, v) ∈ P
    If e = (u, v) is a forward edge then
        increase f(e) in G by b
    Else ((u, v) is a backward edge, and let e = (v, u))
        decrease f(e) in G by b
    Endif
Endfor
Return(f )
```

It was purely to be able to perform this operation that we defined the residual graph.

To reflect the importance of augment, one often refers to any s-t path in the residual graph as an augmenting path.

The result of augment(f, P) is a new flow f' in G, obtained by increasing and decreasing the flow values on edges of P.

Let us first verify that f' is indeed a flow.

Lemma 1 f' is a flow in G.

Proof. We must verify the capacity and conservation conditions for *f'* .

Since *f'* differs from *f* only on edges of *P*, we need to check the capacity conditions only on these edges.

Thus, let (*u*, *v*) be an edge of *P*.

Informally, the capacity condition continues to hold because if $e = (u, v)$ is a forward edge, we specifically avoided increasing the flow on *e* above c_e;

and if (*u*, *v*) is a backward edge arising from edge $e = (v, u) \in E$, we specifically avoided decreasing the flow on *e* below 0.

More concretely, note that *bottleneck(P, f)* $\leq$ the residual capacity of *(u, v)*.

If $e = (u, v)$ is a forward edge, then its residual capacity is $c_e - f(e)$;
thus we have
$f'(e) = f(e) + \text{bottleneck}(P, f) \geq f(e) \geq 0$
$f'(e) = f(e) + \text{bottleneck}(P, f) \leq f(e) + (c_e - f(e)) = c_e$,
so the capacity condition holds.

If (u, v) is a backward edge arising from edge $e = (v, u) \in E$, then its residual capacity is $f(e)$, so we have
$f'(e) = f(e) - \text{bottleneck}(P, f) \leq f(e) \leq f(e)$
$f'(e) = f(e) - \text{bottleneck}(P, f) \geq f(e) - f(e) = 0$,
and again the capacity condition holds.

Next, we need to check the conservation condition at each internal node that lies on the path *P*.

Let *v* be such a node; we can verify that the change in the amount of flow entering *v* is the same as the change in the amount of flow exiting *v*;

since *f* satisfied the conservation condition at *v*, so must *f'*.

Technically, there are 4 cases to check, depending on whether the edge of *P* that enters *v* is a forward or backward edge, and whether the edge of *P* that exits *v* is a forward or backward edge.

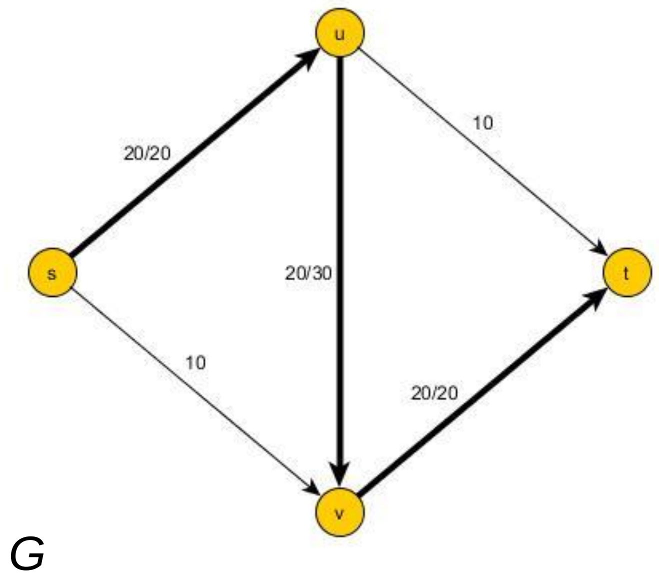
However, each of these cases is easily worked out, and we leave them as an exercise.

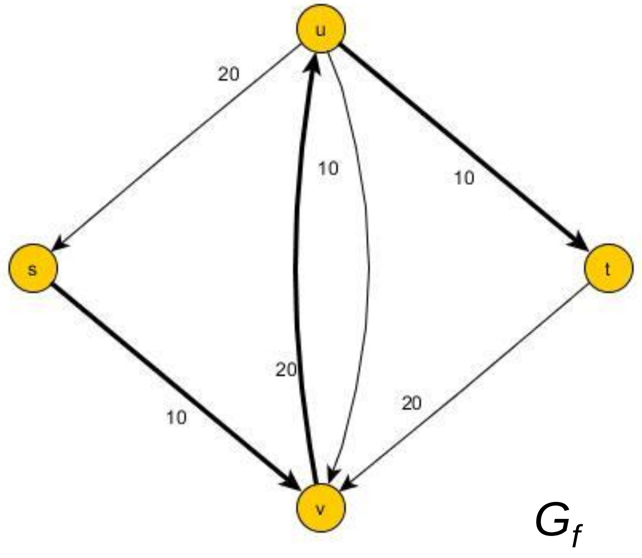
This augmentation operation captures the type of forward and backward pushing of flow that we discussed earlier.

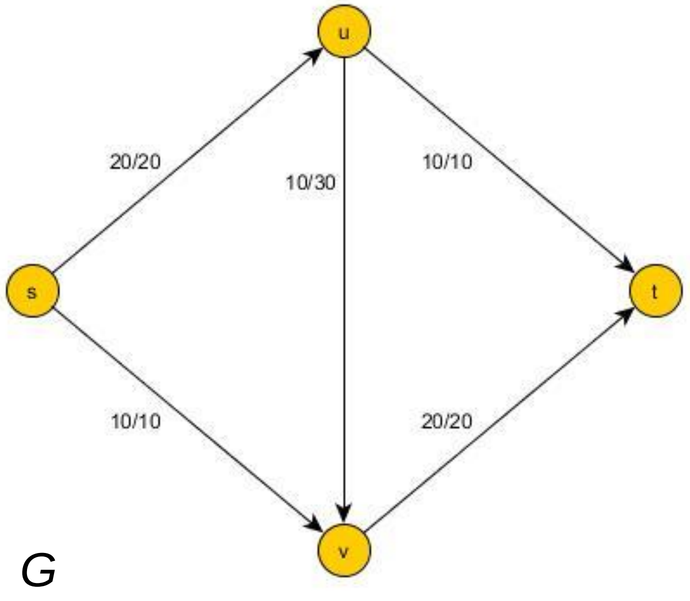
Let’s now consider the following algorithm to compute an s-t flow in G.

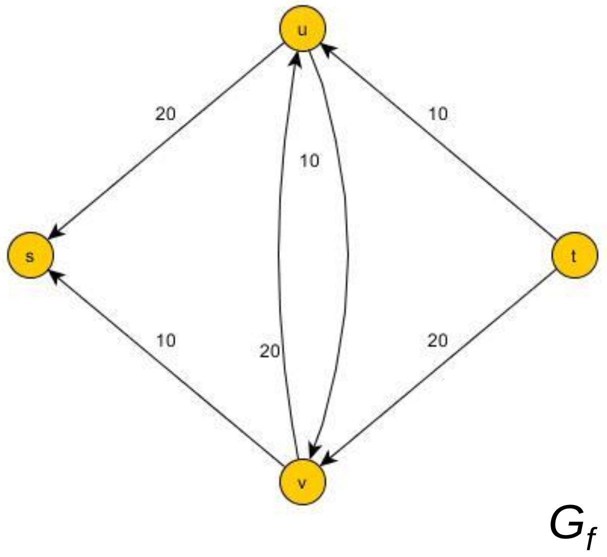
```
Max-Flow
Initially $f(e) = 0$ for all $e$ in $G$
While there is an $s$-$t$ path in the residual graph $G_f$
    Let $P$ be a simple $s$-$t$ path in $G_f$
    $f' = augment(f, P)$
    Update $f$ to be $f'$
    Update the residual graph $G_f$ to be $G_{f'}$
Endwhile
Return $f$
```

It is called the Ford-Fulkerson Algorithm, after the 2 researchers who developed it in 1956.


u
s
t
v
20/20
10
20/30
10
20/20
G


u
s
t
v
20
10
10
20
10
20
G_f


u
s
t
v
20/20
10/30
10/10
10/10
20/20
G


u
s
t
v
20
10
10
10
20
20
G_f

The Ford-Fulkerson Algorithm is really quite simple.
What is not at all clear is:

- whether its central **While** loop terminates, and
- whether the flow returned is a maximum flow.

The answers to both of these questions turn out to be fairly subtle.

Analyzing the Algorithm: Termination and Running Time

First we consider some properties that the algorithm maintains by induction on the number of iterations of the **While** loop, relying on our assumption that all capacities are integers.

Lemma 2 At every intermediate stage of the Ford-Fulkerson Algorithm, the flow values {*f(e)*} and the residual capacities in G_f are integers*.*

Proof. The statement is clearly true before any iterations of the **While** loop.

Now suppose it is true after *j* iterations.

Then, since all residual capacities in G_f are integers, the value bottleneck(*P, f*) for the augmenting path found in iteration $j + 1$ will be an integer.

Thus the flow *f* will have integer values, and hence so will the capacities of the new residual graph.

We can use this property to prove that the Ford-Fulkerson terminates.

We will look for a measure of progress that will imply termination.

First we show that the flow value strictly increases when we apply an augmentation.

Lemma 3 Let f be a flow in G, and let P be a simple s-t path in G_f .
Then $v(f') = v(f\,) + bottleneck(P, f\,)$;
and since $bottleneck(P, f) > 0$, we have $v(f') > v(f)$.

Proof. The first edge e of P must be an edge out of s in the residual graph G_f ; and since the path is simple, it does not visit s again.

Since G has no edges entering s, the edge e must be a forward edge.

We increase the flow on this edge by $bottleneck(P, f\,)$, and we do not change the flow on any other edge incident to s.

Therefore the value of f' exceeds the value of f by $bottleneck(P, f\,)$.

We need one more observation to prove termination:

We need to be able to bound the maximum possible flow value.

Here's one upper bound:

If all the edges out of *s* could be completely saturated with flow, the value of the flow would be

$$\sum_{e \text{ out of } s} c_e .$$

Let us denote this sum *C* .

Thus we have *v(f) ≤ C* for all *s*-*t* flows *f* .

(*C* may be a huge overestimate of the maximum value of a flow in *G*, but it's handy for us as a finite, simply stated bound.)

Using Lemma 3, we can now prove termination.

Lemma 4 Suppose, as above, that all capacities in the flow network *G* are integers.

Then the Ford-Fulkerson Algorithm terminates in $\leq C$ iterations of the **While** loop.

Proof. We noted above that no flow in *G* can have value $> C$, due to the capacity condition on the edges leaving *s*.

Now, by Lemma 3, the value of the flow maintained by the Ford-Fulkerson Algorithm increases in each iteration;

so by Lemma 2, it increases by at least 1 in each iteration.

Since it starts with the value 0, and cannot go higher than *C*, the **While** loop in the Ford-Fulkerson Algorithm can run for $\leq C$ iterations.

Next we consider the running time of the Ford-Fulkerson Algorithm.

Let n denote the number of nodes in G, and m denote the number of edges in G.

We have assumed that all nodes have at least 1 incident edge, hence $m \geq n/2$, and so we can use $O(m + n) = O(m)$ to simplify the bounds.

Lemma 5 Suppose, as above, that all capacities in the flow network G are integers.

Then the Ford-Fulkerson Algorithm can be implemented to run in $O(mC)$ time.

Proof. We know from Lemma 4 that the algorithm terminates in at most C iterations of the **While** loop.

We therefore consider the amount of work involved in 1 iteration when the current flow is f .

The residual graph G_f has at most $2m$ edges, since each edge of G gives rise to ≤ 2 edges in the residual graph.

We will maintain G_f using an adjacency list representation;

we will have 2 linked lists for each node v, one containing the edges entering v, and one containing the edges leaving v.

a) To find an *s*-*t* path in G_f , we can use breadth-first search or depth-first search, which run in *O(m + n) time;*

by our assumption that $m \geq n/2$, $O(m + n)$ is the same as $O(m)$.

b) The procedure *augment*(*f*, *P*) takes time $O(n)$, as the path *P* has at most $n - 1$ edges.

c) Given the new flow *f,* we can build the new residual graph in $O(m)$ time:

For each edge *e* of *G*, we construct the correct forward and backward edges in G_f .

Maximum Flows and Minimum Cuts in a Network

We now continue with the analysis of the Ford-Fulkerson Algorithm.

Our next goal is to show that the flow that is returned by the Ford-Fulkerson Algorithm has the maximum possible value of any flow in *G*.

Analyzing the Algorithm: Flows and Cuts

We have already seen one upper bound: the value *v*(*f*) of any *s*-*t*-flow *f* is at most

$$C = \sum_{e \text{ out of } s} c_e.$$

Sometimes this bound is useful, but sometimes it is very weak. We now use the notion of a cut to develop a much more general means of placing upper bounds on the maximum-flow value.

Consider dividing the nodes of the graph into 2 sets, *A* and *B*, so that $s \in A$ and $t \in B$.

Any such division places an upper bound on the maximum possible flow value, since all the flow must cross from *A* to *B* somewhere.

Formally, we say that an *s-t* cut is a partition (*A, B*) of the vertex set *V*, so that $s \in A$ and $t \in B$.

The capacity of a cut (*A, B*), which we will denote *c(A, B*), is simply the sum of the capacities of all edges out of *A*:

$$c(A, B) = \sum_{e \text{ out of } A} c_e.$$

Cuts turn out to provide very natural upper bounds on the values of flows.

We make this precise via a sequence of facts.

Lemma 6 Let f be any s-t flow, and (A, B) any s-t cut. *Then* $v(f) = f^{out}(A) - f^{in}(A)$.

This statement is actually much stronger than a simple upper bound.

It says that by watching the amount of flow f sends across a cut, we can exactly measure the flow value:

It is the total amount that leaves A, minus the amount that “swirls back” into A.

This makes sense intuitively, although the proof needs a little manipulation of sums.

Proof. By definition $v(f) = f^{out}(s)$.

By assumption we have $f^{in}(s) = 0$, as the source s has no entering edges, so we can write $v(f) = f^{out}(s) - f^{in}(s)$.

Since every node v in A other than s is internal, we know that $f^{out}(v) - f^{in}(v) = 0$ for all such nodes.

Thus

$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v))$$

since the only term in this sum that is nonzero is the one in which $v = s$.

Let's try to rewrite the sum on the right as follows.
• If an edge e has both ends in A, then $f(e)$ appears once in the sum with a "+" and once with a "−", and hence these 2 terms cancel out.
• If e has only its tail in A, then $f(e)$ appears just once in the sum, with a "+".
• If e has only its head in A, then $f(e)$ also appears just once in the sum, with a "−".
• If e has neither end in A, then $f(e)$ doesn't appear in the sum at all.

In view of this, we have

$$\sum_{v \in A} f^{\text{out}}(v) - f^{\text{in}}(v) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

$$= f^{\text{out}}(A) - f^{\text{in}}(A).$$

Putting together these 2 equations, we have the statement of Lemma 6.

If $A = \{s\}$, then $f^{\text{out}}(A) = f^{\text{out}}(s)$, and $f^{\text{in}}(A) = 0$ as there are no edges entering the source by assumption.

So the statement for this set $A = \{s\}$ is exactly the definition of the flow value $v(f)$.

Note that if (A, B) is a cut, then the edges into B are precisely the edges out of A.
Similarly, the edges out of B are precisely the edges into A.
Thus we have $f^{out}(A) = f^{in}(B)$ and
$f^{in}(A) = f^{out}(B)$,
just by comparing the definitions for these 2 expressions.

So we can rephrase Lemma 6 in the following way.
Lemma 7 Let f be any s-t flow, and (A, B) any s-t cut.
Then $v(f) = f^{in}(B) - f^{out}(B)$.
If we set $A = V - \{t\}$ and $B = \{t\}$ in Lemma 7,
we have $v(f) = f^{in}(B) - f^{out}(B) = f^{in}(t) - f^{out}(t)$.

By our assumption the sink t has no leaving edges, so we have $f^{out}(t) = 0$.
This says that we could have originally defined the value of a flow equally well in terms of the sink t:
It is $f^{in}(t)$, the amount of flow arriving at the sink.

A very useful consequence of Lemma 6 is the following upper bound.

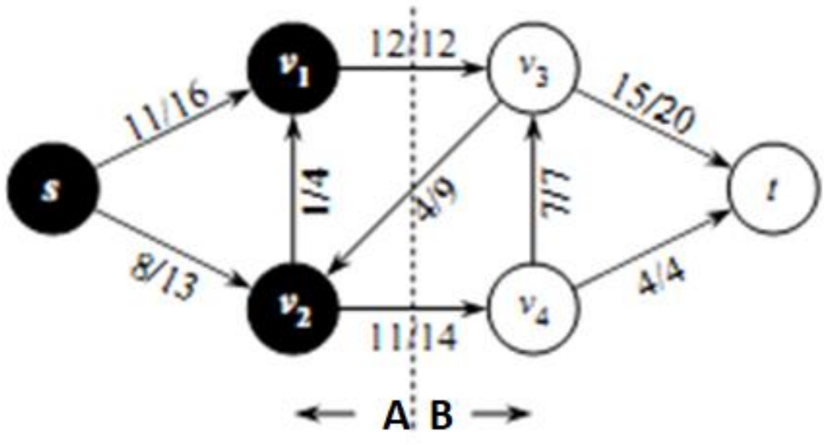
Lemma 8 Let *f* be any *s*-*t* flow, and (*A*, *B*) any *s*-*t* cut.
Then *v(f) ≤ c(A, B)*.

Proof.

$$\begin{aligned} \nu(f) &= f^{\text{out}}(A) - f^{\text{in}}(A) \\ &\le f^{\text{out}}(A) \\ &= \sum_{e \text{ out of } A} f(e) \\ &\le \sum_{e \text{ out of } A} c_e \\ &= c(A, B). \end{aligned}$$

Here the first line is simply Lemma 6;
we pass from the first to the second since $f^{\text{in}}(A) \ge 0$,
and we pass from the third to the fourth by applying the capacity conditions to each term of the sum.

Example of cut and cut capacity



A cut (A, B) in the flow network where $A = \{s, v_1, v_2\}$ and $B = \{v_3, v_4, t\}$.
The vertices in A are black, and the vertices in B are white.
The flow across (A, B) is $f(A, B) = f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) = 12 + 11 - 4 = 19$,
and the capacity is $c(A, B) = c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26$.

In a sense, Lemma 8 looks weaker than Lemma 6, since it is only an inequality rather than an equality.

However, it will be extremely useful, since its right-hand side is independent of any particular flow *f* .

What Lemma 8 says is that the value of every flow is upper-bounded by the capacity of every cut.

In other words, if we exhibit any *s*-*t* cut in *G* of some value c^*, we know immediately by Lemma 8 that there cannot be an *s*-*t* flow in *G* of value $> c^*$.

Conversely, if we exhibit any *s*-*t* flow in *G* of some value v^*, we know immediately by Lemma 8 that there cannot be an *s*-*t* cut in *G* of value $< v^*$.

Analyzing the Algorithm: Max-Flow Equals Min-Cut

Let *f* denote the flow that is returned by the Ford-Fulkerson Algorithm.

We want to show that *f* has the maximum possible value of any flow in *G*, and we do this by the method discussed above:

We exhibit an *s*-*t* cut (A^*, B^*) for which $v(f) = c(A^*, B^*)$.

This immediately establishes that *f* has the maximum value of any flow, and that (A^*, B^*) has the minimum capacity of any *s*-*t* cut.

The Ford-Fulkerson Algorithm terminates when the flow *f* has no *s*-*t* path in the residual graph G_f .

This turns out to be the only property needed for proving its maximality.

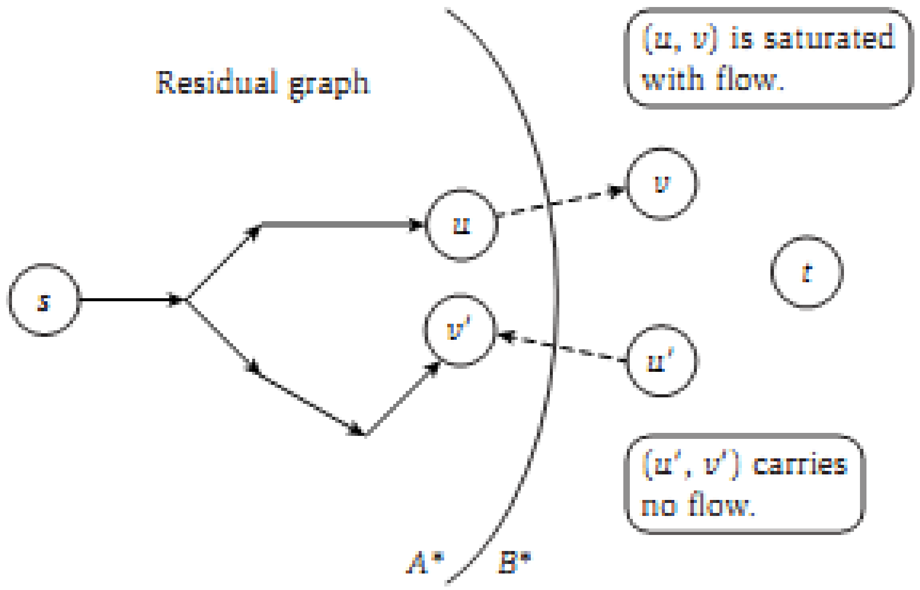
Lemma 9 If f is an *s-t-flow* such that there is no s-t path in the residual graph G_f, then there is an s-t cut (A^*, B^*) in G for which $v(f) = c(A^*, B^*)$.

Consequently, f has the maximum value of any flow in G, and (A^*, B^*) has the minimum capacity of any s-t cut in G.

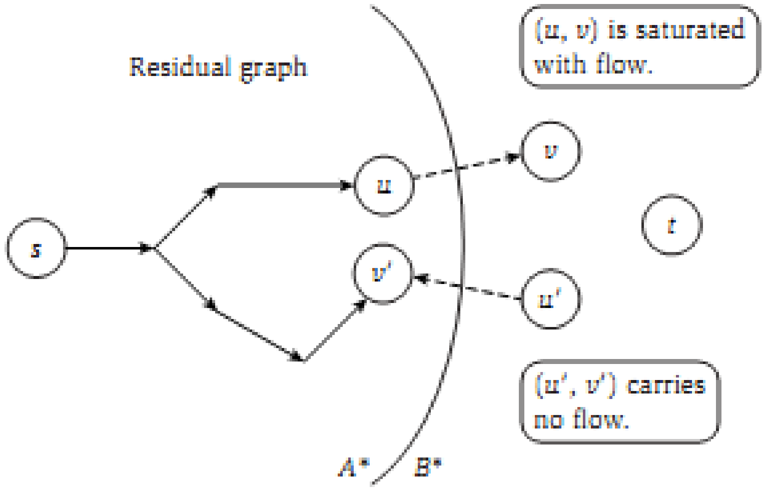
Proof. The statement claims the existence of a cut satisfying a certain desirable property; thus we must now identify such a cut.

To this end, let A^* denote the set of all nodes v in G for which there is an s-v path in G_f .

Let B^* denote the set of all other nodes: $B^* = V - A^*$.



First we establish that (*A**, *B**) is indeed an *s*-*t* cut.
It is clearly a partition of *V*.
The source *s* belongs to *A** since there is always a path from *s* to *s*. Moreover, $t \notin A^*$ by the assumption that there is no *s*-*t* path in the residual graph; hence $t \in B^*$ as desired.



Next, suppose that $e = (u, v)$ is an edge in G for which $u \in A^*$ and $v \in B^*$, as shown upward.

We claim that $f(e) = c_e$.

For if not, e would be a forward edge in the residual graph G_f , and since $u \in A^*$, there is an s-u path in G_f ;

appending e to this path, we would obtain an s-v path in G_f, contradicting our assumption that $v \in B^*$.

Now suppose that $e' = (u', v')$ is an edge in G for which $u' \in B^*$ and $v' \in A^*$.

We claim that $f(e') = 0$.

For if not, e' would give rise to a backward edge $e = (v', u')$ in the residual graph G_f , and since $v \in A^*$, there is an s-v path in G_f ; appending e to this path, we would obtain an s-u path in G_f , contradicting our assumption that $u \in B^*$.

So all edges out of A^* are completely saturated with flow, while all edges into A^* are completely unused.

We can now use Lemma 6 to reach the desired conclusion:

$$\begin{aligned}
\nu(f) &= f^{\text{out}}(A^*) - f^{\text{in}}(A^*) \\
&= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\
&= \sum_{e \text{ out of } A^*} c_e - 0 \\
&= c(A^*, B^*). \quad \blacksquare
\end{aligned}$$

Now we can see why the 2 types of residual edges—forward and backward—are crucial in analyzing the two terms in the expression from Lemma 6.

Given that the Ford-Fulkerson Algorithm terminates when there is no *s*-*t* path in the residual graph, Lemma 6 immediately implies its optimality.

Lemma 10 The flow *f* returned by the Ford-Fulkerson Algorithm is a maximum flow.

We also observe that our algorithm can easily be extended to compute a minimum *s*-*t* cut ($A*$, $B*$), as follows.

Lemma 11 Given a flow *f* of maximum value, we can compute an *s*-*t* cut of minimum capacity in $O(m)$ time.

Proof. We simply follow the construction in the proof of Lemma 9.

We construct the residual graph G_f, and perform breadth-first search or depth-first search to determine the set A^* of all nodes that *s* can reach.

We then define $B* = V - A^*$, and return the cut (A^*, B^*).

Note that there can be many minimum-capacity cuts in a graph *G*;

the procedure in the proof of Lemma 11 is simply finding a particular one of these cuts, starting from a maximum flow *f* .

As a bonus, we have obtained the following striking fact through the analysis of the algorithm.

Lemma 12 In every flow network, there is a flow *f* and a cut *(A, B)* so that $v(f) = c(A, B)$.

The point is that *f* in Lemma 12 must be a maximum *s*-*t* flow; for if there were a flow *f* of greater value, the value of *f* would exceed the capacity of (*A*, *B*), and this would contradict Lemma 8.

Similarly, it follows that (*A, B*) in Lemma 12 is a minimum cut—no other cut can have smaller capacity—for if there were a cut (*A, B*) of smaller capacity, it would be less than the value of *f*, and this again would contradict Lemma 8.

Due to these implications, Lemma 12 is often called the Max-Flow Min-Cut Theorem , and is phrased as follows.

In every flow network, the maximum value of an *s*-*t* flow is equal to the minimum capacity of an *s*-*t* cut.

Choosing Good Augmenting Paths

Let us discuss now how to select augmenting paths so as to avoid the potential bad behavior of the algorithm.

Previously, we saw that any way of choosing an augmenting path increases the value of the flow, and this led to a bound of *C* on the number of augmentations, where

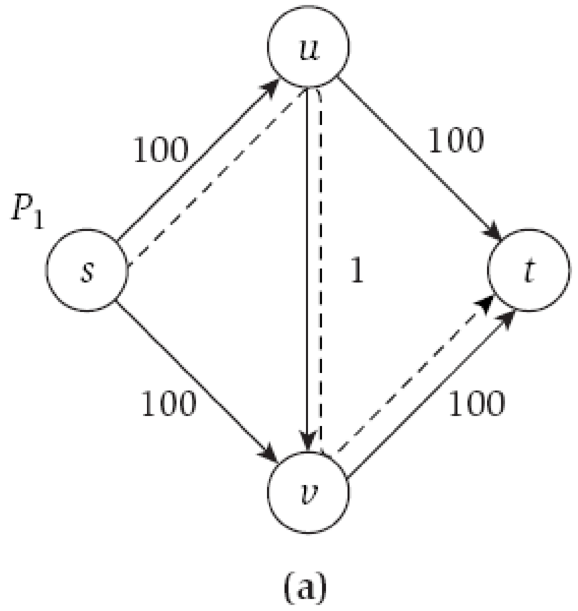
.

$$C = \sum_{e \text{ out of } s} c_e.$$

When *C* is not very large, this can be a reasonable bound; however, it is very weak when *C* is large.

To get a sense for how bad this bound can be, consider the previous flow network but this time assume the capacities are as follows.

The edges (*s, v*), (*s, u*), (*v, t*) and (u*, t*) have capacity 100, and the edge (*u, v*) has capacity 1.



(a)

It is easy to see that the maximum flow has value 200, and has *f*(*e*) = 100 for the edges (*s, v*), (*s, u*), (*v, t*) and (*u, t*) and value 0 on the edge (*u, v*).

This flow can be obtained by a sequence of 2 augmentations,

using the paths of nodes *s*, *u*, *t* and path *s*, *v*, *t*

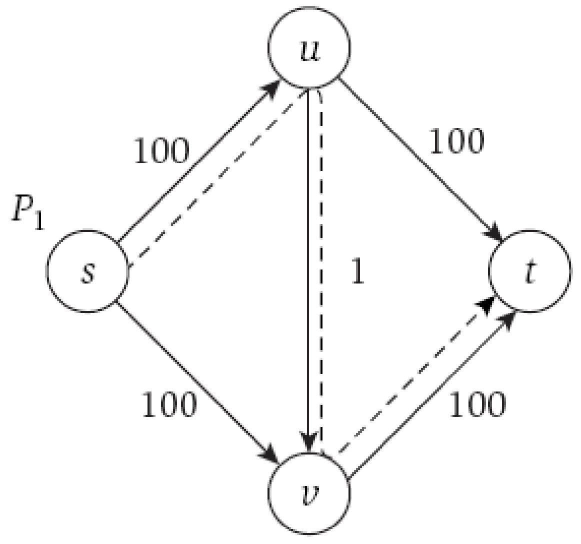
But consider how bad the Ford-Fulkerson Algorithm can be with pathological choices for the augmenting paths.

Suppose we start with augmenting path *P_1* of nodes *s, u, v, t* in this order (as shown on the right).

This path has bottleneck$(P_1, f) = 1$.

After this augmentation, we have

$f(e) = 1$ on the edge $e = (u, v)$, so the reverse edge is in the residual graph.



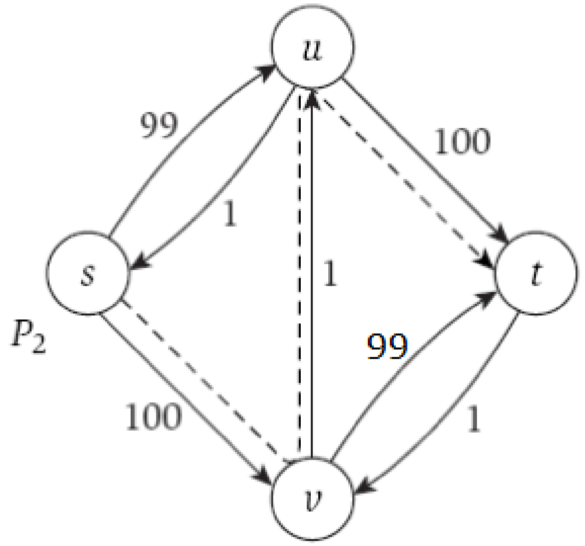
(a)

For the next augmenting path, we choose the path P_2 of the nodes *s*, *v*, *u*, *t* in this order.

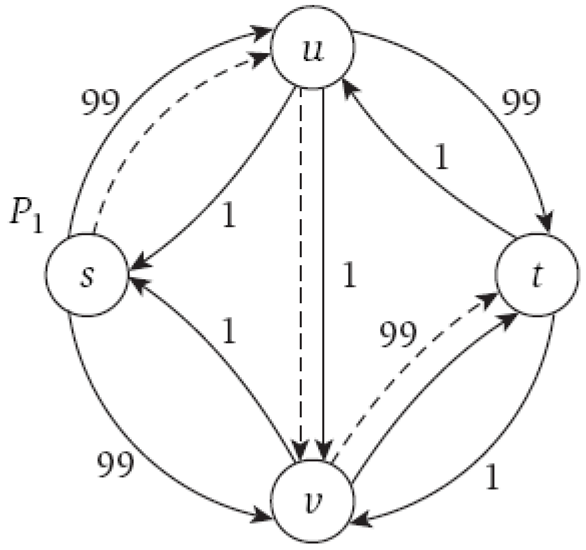
In this second augmentation, we get bottleneck(P_2, f) = 1 as well.

After this second augmentation, we have $f(e) = 0$ for the edge $e = (u, v)$, so the edge is again in the residual graph.

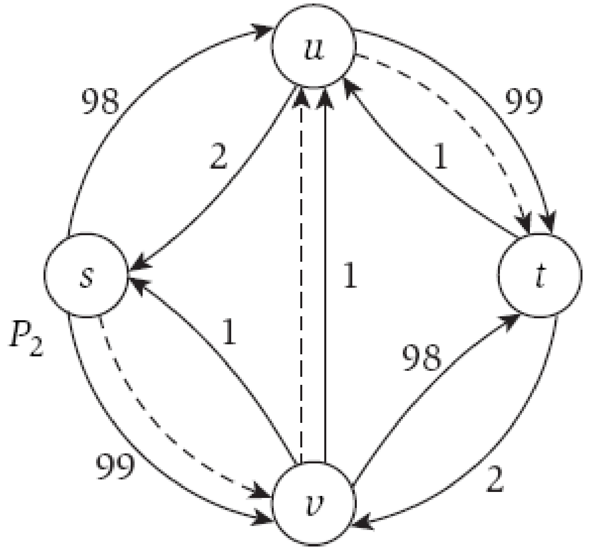
Suppose we alternate between choosing P_1 and P_2 for augmentation.



(b)



(c)



(d)

In this case, each augmentation will have 1 as the bottleneck capacity, and it will take 200 augmentations to get the desired flow of value 200.

This is exactly the bound we proved in Lemma 4, since *C* = 200 in this example.

The Edmonds-Karp algorithm

We can improve the bound on FORD-FULKERSON by finding the augmenting path P with a breadth-first search.

That is, we choose the augmenting path as a shortest path from s to t in the residual network, where each edge has unit distance (weight).

We call the Ford-Fulkerson method so implemented the Edmonds-Karp algorithm.

We now claim that the Edmonds-Karp algorithm runs in $O(VE^2)$ time.

Theorem 13
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Because it is possible to implement each iteration of FORD-FULKERSON in $O(E)$ time when we find the augmenting path by breadth-first search, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$.