In these next two lectures we are going to talk about an important algorithmic problem called the *Network Flow Problem*. Network flow is important because it can be used to express a wide variety of different kinds of problems. So, by developing good algorithms for solving network flow, we immediately will get algorithms for solving many other problems as well. In Operations Research there are entire courses devoted to network flow and its variants. Topics in today's lecture include:

- The definition of the network flow problem
- The basic Ford-Fulkerson algorithm
- The maxflow-mincut theorem
- The bipartite matching problem

1 The Network Flow Problem

We begin with a definition of the problem. We are given a simple directed graph G, a source node s, and a sink node t. Recall that a simple graph contains no parallel edges (a pair of edges with the same endpoints that point in the same direction) and no loops (an edge from a vertex to itself). We assume that every vertex is the graph is reachable from s, and that t is reachable from every vertex of the graph, so there are no "useless" vertices. Each edge e in G has an associated non-negative *capacity* $c(e)$, where for all non-edges it is implicitly assumed that the capacity is 0. For instance, imagine we want to route message traffic from the source to the sink, and the capacities tell us how much bandwidth we're allowed on each edge. For example, consider the graph in Figure 1 below.

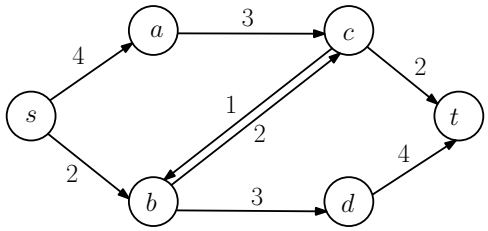


Figure 1: A network flow graph.

Our goal is to push as much *flow* as possible from s to t in the graph. The rules are that no edge can have flow exceeding its capacity, and for any vertex except for s and t, the flow *in* to the vertex must equal the flow *out* from the vertex. That is,

Capacity constraint: On any edge e we have $0 \leq f(e) \leq c(e)$.

Flow conservation: For any vertex $v \notin \{s, t\}$, flow in equals flow out: $\sum_u f(u, v) = \sum_u f(v, u)$.

A flow that satisfies these constraints is called a *feasible flow*. Subject to these constraints, we want to maximize the net flow from s to t. We can measure this formally by the quantity

$$|f| = \sum_u f(s, u) - \sum_u f(u, s).$$

Note that what we are measuring here is the net flow coming out of s. It is straightforward to prove, due to conservation, that this is equal to the net flow coming into t.

E.g., in the above graph, what is the maximum flow from s to t? Answer: 5. Using "capacity[flow]" notation, the positive flow looks as in Figure 2. Note that the flow can split and rejoin itself.

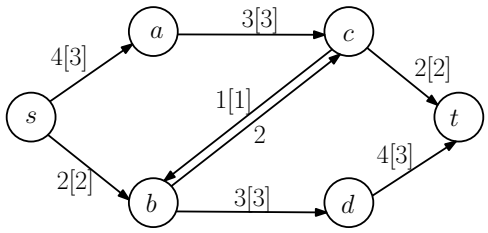


Figure 2: A network flow graph with positive flow shown using "capacity[flow]" notation.

An alternate formulation The way that we have defined a flow f above is sometimes referred to as the *gross flow* formulation, where we measure the non-negative amount of flow on each edge e as $0 \leq f(e) \leq c(e)$. An alternative formulation that is sometimes used is called the *net flow* formulation. In this world, we allow the flow value $f(u, v)$ to be negative to indicate that a flow is flowing in the *opposite direction* of a given edge. This means that for all $(u, v) \in E$, we define $f(v, u) = -f(u, v)$. This property is called *skew-symmetry*. Please note that this is not a different problem to the one above, its just a slightly different way of defining what a flow is. All of our theorems and algorithms work exactly the same in both versions, all that changes are some minor details in some definitions and proofs.

In this formulation, the conservation constraint can also be simplified to $\forall v \notin \{s, t\}, \sum_u f(u, v) = 0$, since the total flow *out* of a node will always be the negative of the total flow *into* a node. Similarly, the value of a flow can then be measured by $|f| = \sum_u f(s, u)$ since all of the incoming flow is now treated as negative flow by the skew symmetry definition.

Unless otherwise stated, I'll write things using the gross flow formulation, since it is more explicit and in my opinion clearer to work with.

1.1 Improving a flow: s-t paths

Suppose we start with the simplest possible flow, the all-zero flow. This flow is clearly feasible since it meets the capacity constraints, and all vertices have flow in equal to flow out (zero!). But for the network in Figure 1, the all-zero flow is clearly not optimal. Can we formally reason about *why* it isn't optimal? One way is to observe that there exists a path from s to t in the graph consisting of edges with available capacity (actually there are many such paths). For example, the path $s \to a \to c \to t$ has at least 2 capacity available, so we could add 2 flow to each of those edges without violating their capacity constraints, and improve the value of the flow. This gives us the flow below in Figure 3.

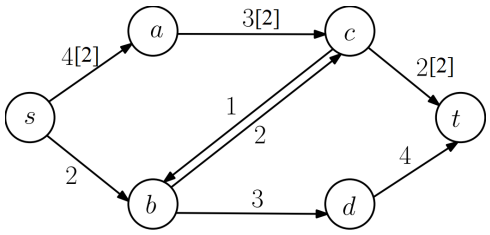


Figure 3: A network flow graph with 2 units of flow added to the path $s \to a \to c \ \to t$.

An important observation we have just made is that if we add a constant amount of flow to all of the edges on a path, we never violate the flow conservation condition since we add the same amount of flow in and flow out to each vertex, except for s and t. As long as we do not violate the capacity of any edge, the resulting flow is therefore still a feasible flow.

This observation leads us to the following idea: A flow is *not optimal* if there exists an s-t path with available capacity. In the above graph, we can observe that the path $s \to b \to d \to t$ has 2 more units of available capacity, and hence add 2 units of flow to the edges. Lastly, we could find the $s \to a \to c \to b \to d \to t$ path has 1 more unit of available capacity, and we would arrive at the maximum flow from Figure 2.

1.2 Certifying optimality of a flow: s-t cuts

We just saw how to convince ourselves that a flow was *not* maximum, but how can you see that the above flow was really maximum? We can't be certain yet that we didn't make a bad decision and send flow down a sub-optimal path. Here's an idea. Notice, this flow saturates the $a \to c$ and $s \to b$ edges, and, if you remove these, you disconnect t from s. In other words, the graph has an "s-t cut" of size 5 (a set of edges of total capacity 5 such that if you remove them, this disconnects the sink from the source). The point is that any unit of flow going from s to t must take up at least 1 unit of capacity in these pipes. So, we know we're optimal.

What we have just argued is that the value of any flow is less than the value of any s-t cut. Since every flow has a value that is at most the maximum flow, we can argue that in general, the maximum s-t flow $\leq$ the capacity of the minimum s-t cut, or

the value of *any* s-t flow $\leq$ maximum s-t flow $\leq$ minimum s-t cut $\leq$ capacity of *any* s-t cut.

An important property of flows, that we will prove as a byproduct of analyzing an algorithm for finding them, is that the maximum s-t flow is in fact *equal to* the capacity of the minimum s-t cut. This is called the *Maxflow-Mincut Theorem*. In fact, the algorithm will find a flow of some value k and a cut of capacity k, which will be proofs that both are optimal!

To describe the algorithm and analysis, it will help to be a bit more formal about a few of these quantities.

Definition 1 *An* s-t **cut** *is a set of edges whose removal disconnects* t *from* s*. Or, formally, a cut is a partition of the vertex set into two pieces* S *and* T *where* $s \in S$ *and* $t \in T$*. (The edges of the cut are then all edges going from* S *to* T*).*

Definition 2 *The* **capacity** *of a cut* (S, T) *is the sum of capacities of edges in the cut. Or, in the formal viewpoint, it is the sum of capacities of all edges going from* S *to* T*.*

$$cap(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

Note importantly this we don't include the edges from T to S in this definition.

Definition 3 *The **net flow** across a cut (S, T) is the sum of flows going from S to T minus the flows going from T to S, or, more formally*[1]

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u).$$

From this definition, we can see that the value of a flow $|f|$ is equivalent to the net flow across the cut $(\{s\}, V \setminus \{s\})$. In fact, using the flow conservation property, it is not hard to show that the net flow across *any* cut is equal to the flow value $|f|$. This should make intuitive sense, because no matter where we cut the graph, there is still the same amount of flow making it from s to t.

1.3 Finding a maximum flow

How can we find a maximum flow and prove it is correct? We should try to tie together the two ideas from above. Here's a very natural strategy: find a path from s to t and push as much flow on it as possible. Then look at the leftover capacities (an important issue will be how exactly we define this, but we will get to it in a minute) and repeat. Continue until there is no longer any path with capacity left to push any additional flow on. Of course, we need to *prove* that this works: that we can't somehow end up at a suboptimal solution by making bad choices along the way. Is this the case for a naive algorithm that simply searches for s-t paths with available capacity and adds flow to them? Unfortunately it is not. Consider the following graph, where the algorithm has decided to add 1 unit of flow to the path $s \to a \to b \to t$.

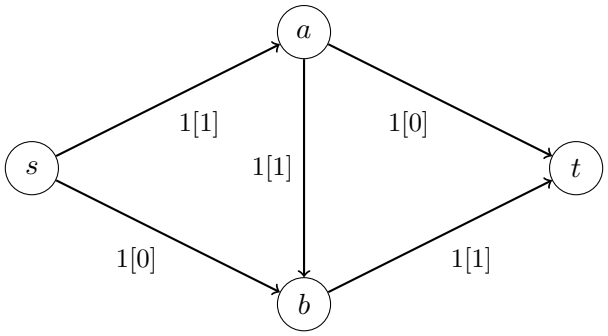


Figure 4: An example of a suboptimal flow in which there are no longer any s-t paths with available capacity but the flow is not maximum.

Are there any s-t paths with available capacity left? No, there are not. But is this flow a maximum flow? Also no, because we could have sent 1 unit of flow from $s \to a \to t$ and 1 unit of flow from $s \to b \to t$ for a value of 2.

The problem with this attempted solution was that sending flow from a to b was a "mistake" that lowered the amount of available capacity in the network. To arrive at an optimal algorithm for maximum flow, we therefore need a way of "undoing" such mistakes. We achieve this by defining

[1] If we were using the *net flow* formulation of flows with skew symmetry, we would drop the second term in this definition because those flows from T to S would be automatically counted as "negative flows" from S to T

the notion of *residual capacity*, which accounts for both the remaining capacity on an edge, but also adds the ability to *undo* bad decisions and redirect a suboptimal flow to a more optimal one. This leads us to the Ford-Fulkerson algorithm.

2 The Ford-Fulkerson algorithm

The Ford-Fulkerson algorithm[2] is simply the following: while there exists an $s \to t$ path P of positive *residual capacity* (defined below), push the maximum possible flow along P. By the way, these paths P are called *augmenting paths*, because you use them to augment the existing flow.

Residual capacity is just the capacity left over given the existing flow and also accounts for the ability to push existing flow back and along a different path, in order to undo previous bad decisions.

Definition 4 *Given a flow f in graph G, the* **residual capacity** $c_f(u, v)$ *is defined as*[3]

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

Definition 5 *Given a flow f in graph G, the* **residual graph** G_f *is the directed graph with all edges of positive residual capacity, each one labeled by its residual capacity. Note: this may include reverse-edges of the original graph G.*

When $(u, v) \in E$, the residual capacity on the edge corresponds to our existing intuitive notion of "remaining/leftover" capacity, it is the amount of additional flow that we could put through an edge before it is full. The second case is the key insight that makes the Ford-Fulkerson algorithm work. If $f(v, u)$ flow has been sent down some edge (v, u), then we are able to *undo* that by sending flow back in the reverse direction (u, v)! For example, if we consider the suboptimal flow in Figure 4, the residual graph looks like the following.

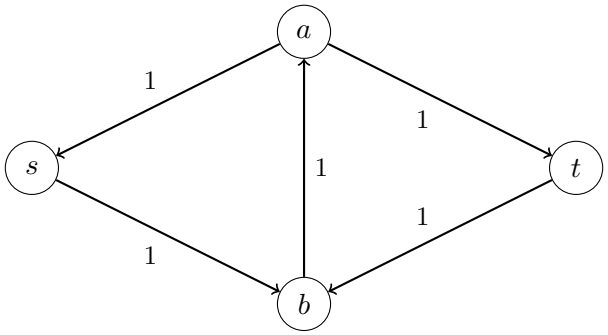


Figure 5: The residual graph corresponding to the flow in Figure 4.

[2]The algorithm is due to Lester R. Ford Jr. and Delbert R. Fulkerson, then at the RAND corporation working on figuring out the capacity of the Soviet rail network. They didn't want to find the max-flow, they wanted the min-cut. As we will soon see, the two problems are inextricably entwined. They presented their algorithm in a RAND report of 1955.

[3]If you prefer the net flow formulation of flows, you can write the residual capacity as just $c_f(u, v) = c(u, v) - f(u, v)$ for all (u, v). Note that by the skew symmetry property since $f(u, v) = -f(v, u)$ then if $(v, u) \notin E$, we get $c_f(u, v) = 0 - -f(u, v) = f(v, u)$ which matches the gross flow definition.

Notice that unlike before, there is now an s-t path with capacity 1! The path $s \to b \to a \to t$ "undoes" the flow that was originally pushed through $a \to b$ and redirects it to $a \to t$, thus improving the flow and making it optimal.

Handing anti-parallel edges The definition of residual capacity becomes a little funny if we suppose that the graph contains anti-parallel edges. Recall that parallel edges are those with the same endpoints u and v that point in the same direction, while anti-parallel edges are those with the same endpoints but point in opposite directions. In this case, it might be the case that both $(u, v) \in E$ and $(v, u) \in E$, so which of the two cases do we use for the residual capacity? The simple answer is that we should use *both*. We have the choice of either combining both into a single edge with the sum of their capacities, or including a pair of parallel edges in the residual graph. Both of these options are fine and will work in theory and practice.

Let's see another example. Consider the graph in Figure 1 and suppose we push two units of flow on the path $s \to b \to c \to t$. We then end up with the following residual graph:

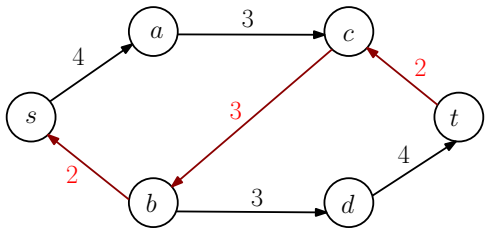


Figure 6: Residual graph resulting from pushing 2 units of flow along the path s-b-c-t in the graph in Figure 1. The red edges denote the changes.

Note that the capacity of 3 on the edge $c \to b$ is due to the edge $c \to b$ in the original graph with capacity 1 plus the residual capacity of 2 due to the flow of 2 going from $b \to c$ in f that we have the ability to undo. We could have alternatively drawn these as two separate edges of capacity 1 and 2, but it is often simpler to combine them so that we don't end up with any parallel edges.

If we continue running Ford-Fulkerson, we see that in this graph the only path we can use to augment the existing flow is the path $s \to a \to c \to b \to d \to t$. Pushing the maximum 3 units on this path we then get the next residual graph, shown in Figure 7.

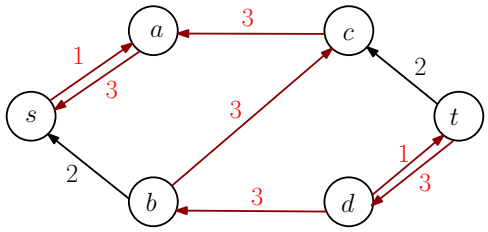


Figure 7: Residual graph resulting from pushing 3 units of flow along the path s-a-c-b-d-t in the graph in Figure 6. Again, the red edges denote the changes.

At this point there is no longer a path from s to t so we are done.

We can think of Ford-Fulkerson as at each step finding a new flow (along the augmenting path) and adding it to the existing flow. The definition of residual capacity ensures that the flow found by Ford-Fulkerson is *legal* (doesn't exceed the capacity constraints in the original graph). We now

need to prove that in fact it is *maximum*. We'll worry about the number of iterations it takes and how to improve that later.

Note that one nice property of the residual graph is that it means that at each step we are left with same type of problem we started with. So, to implement Ford-Fulkerson, we can use any black-box path-finding method (e.g., DFS).

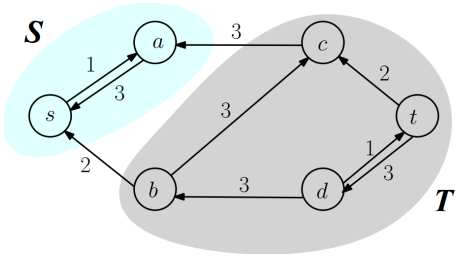
2.1 The Analysis

For now, let us assume that all the capacities are integers. If the maximum flow value is F, then the algorithm makes at most F iterations, since each iteration pushes at least one more unit of flow from s to t. We can implement each iteration in time $O(m+n)$ using DFS. Since we assume the graph is simple, $m \geq n-1$, so this is $O(m)$, so we get the following result.

Theorem 6 *If the given graph G has integer capacities, Ford-Fulkerson terminates in time $O(mF)$ where F is the value of the maximum s-t flow.*

Theorem 7 *When it terminates, the Ford-Fulkerson algorithm outputs a flow whose value is equal to the minimum cut of the graph.*

Proof: Let's look at the final residual graph. Since Ford-Fulkerson loops until there is no longer a path from s to t, this graph must have s and t disconnected, otherwise the algorithm would keep looping. Let S be the set of vertices that are still reachable from s in the residual graph, and let T be the rest of the vertices. It must be the case that $t \in T$ since otherwise s would be connected to t, so this is a valid s-t cut. Let $c = \text{cap}(S,T)$, the capacity of the cut in the *original* graph. From our earlier discussion, we know that $|f| = f(S,T) \leq c$.



The claim is that we in fact *did* find a flow of value c (which therefore implies it is maximum). Consider all of the edges of G (the original graph) that cross the cut in either direction. We consider both cases:

1. Consider edges in G that cross the cut in the $S \to T$ direction. We claim that all of these edges are at maximum capacity (the technical term is *saturated*). Suppose for the sake of contradiction that there was an edge crossing from S to T that was not saturated. Then, by definition of the residual capacity, the residual graph would contain an edge with positive residual capacity from S to T, but this contradicts the fact that T contains the vertices that we can not reach in the residual graph, since we would be able to use this edge to reach a vertex in T. Therefore, we can conclude that every edge crossing the cut from S to T is saturated.

2. Consider edges in G that cross the cut in the T to S direction. We claim that all such edges are *empty* (their flow is zero). Again, suppose for the sake of contradiction that this was not true and there is a non-zero flow on an edge going from T to S. Then by the definition of the

residual capacity, there would be a reverse edge with positive residual capacity going from S to T. Once again, this is a contradiction because this would imply that there is a way to reach a vertex in T in the residual graph. Therefore, every edge crossing from T to S is empty.

Therefore, we have for every edge crossing the cut from S to T that $f(u,v) = c(u,v)$, and for every edge crossing from T to S that $f(u,v) = 0$, so the *net flow* across the cut is equal to the capacity of the cut c. Since every flow has a value that is at most the capacity of the minimum cut, this cut must in fact be the minimum cut, and the value of the flow is equal to it. ■

Notice that in the above argument we actually proved the nonobvious *maxflow-mincut* theorem:

Theorem 8 *In any graph G, for any two vertices s and t, the maximum flow from s to t equals the capacity of the minimum (s,t)-cut.*

We have also proven the *integral-flow theorem*: if all capacities are integers, then there is a maximum flow in which all flows are integers. This seems obvious, but it turns out to have some nice and non-obvious implications.

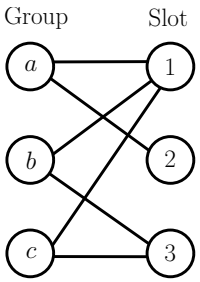
> Technically, we just proved Theorem 8 only for integer capacities. What if the capacities are not integers? Firstly, if the capacities are rationals, then choose the smallest integer N such that $N \cdot c(u,v)$ is an integer for all edges (u,v). It is easy to see that at each step we send at least $1/N$ amount of flow, and hence the number of iterations is at most NF, where F is the value of the maximum s-t flow. (One can argue this by observing that scaling up all capacities by N will make all capacities integers, whence we can apply our above argument.) And hence we get Theorem 8 for rational capacities as well.
>
> What if the capacities are irrational? In this case Ford-Fulkerson may not terminate. And the solution it converges to (in the limit) may not even be the max-flow! (See here[4], here[5].) But the maxflow-mincut Theorem 8 still holds, even with irrational capacities. There are several ways to prove this; here's one. Suppose not, and suppose there is some flow network with the maxflow being $\epsilon > 0$ smaller than the mincut. Choose integer N such that $\frac{1}{N} \leq \frac{\epsilon}{2m}$, and round all capacities down to the nearest integer multiple of $1/N$. The mincut with these new edge capacities may have fallen by $m/N \leq \epsilon/2$, and the maxflow could be the same as the original flow network, but still there would be a gap of $\epsilon/2$ between maxflow and mincut in this rational-capacity network. But this is not possible, because used Ford-Fulkerson to prove maxflow-mincut for rational capacities in the previous paragraph.
>
> Alternatively, in the next lecture we'll see that if we modify Ford-Fulkerson to always choose an augmenting path with the fewest edges on it, it's guaranteed to terminate. This proves the max-flow min-cut theorem for arbitrary capacities.

In the next lecture we will look at methods for reducing the number of iterations the algorithm can take. For now, let's see how we can use an algorithm for the max flow problem to solve other problems as well: that is, how we can *reduce* other problems to the one we now know how to solve.

3 Bipartite Matching

Say we wanted to be more sophisticated about assigning groups to homework presentation slots. We could ask each group to list the slots acceptable to them, and then write this as a bipartite graph by drawing an edge between a group and a slot if that slot is acceptable to that group. For example:

[4] https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/07DemoFordFulkersonPathological.pdf
[5] http://www.cs.huji.ac.il/ nati/PAPERS/ALGORITHMS/zwick.pdf



This is an example of a **bipartite graph**: a graph with two sides L and R such that all edges go between L and R. A **matching** is a set of edges with no endpoints in common. What we want here in assigning groups to time slots is a **perfect matching**: a matching that connects every point in L with a point in R. For example, what is a perfect matching in the bipartite graph above?

More generally (say there is no perfect matching) we want a **maximum matching**: a matching with the maximum possible number of edges. We can solve this as follows:

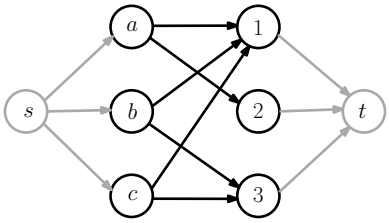
Bipartite Matching:

1. Set up a fake "start" node s connected to all vertices in L. Connect all vertices in R to a fake "sink" node t. Orient all edges left-to-right and give each a capacity of 1.
2. Find a max flow from s to t using Ford-Fulkerson.
3. Output the edges between L and R containing nonzero flow as the desired matching.

This finds a legal matching because edges from R to t have capacity 1, so the flow can't use two edges *into* the same node, and similarly the edges from s to L have capacity 1, so you can't have flow on two edges *leaving* the same node in L. It's a *maximum* matching because any matching gives you a flow of the same value: just connect s to the heads of those edges and connect the tails of those edges to t. (So if there was a better matching, we wouldn't be at a maximum flow).

What about the number of iterations of path-finding? This is at most the number of edges in the matching since each augmenting path gives us one new edge.

Let's run the algorithm on the above example. We first build this flow network.



Then we use Ford-Fulkerson. Say we start by pushing flow on s-a-1-t and s-c-3-t, thereby matching a to 1 and c to 3. These are bad choices, since with these choices b cannot be matched. But the augmenting path s-b-1-a-2-t *automatically* undoes them as it improves the flow!

Matchings come up in many different problems like matching up suppliers to customers, or cell-phones to cell-stations when you have overlapping cells. They are also a basic part of other algorithmic problems.