CIS 320: Algorithms	October 20 and 25, 2021
Lecture 13-14	
Lecturer: Aaron Roth	*Scribe: Aaron Roth*

Max Flow I and II

In the next two lectures, we will revisit the problem we considered in the first lecture: computing max-weight matchings in bipartite graphs. But in order to get there, we will solve a significantly more general problem that can be used to compute max-weight matchings along with a number of other things.

A maximum *flow* problem will also be defined on a directed graph (not necessarily bipartite), and we will intuitively view the edges e in the graph as "pipes" or "wires" that are used for transporting something (e.g. water or information), but that have some capacity c_e. We will have a determined *source* vertex s from which traffic or "flow" originates, and we will have a determined *destination* vertex t which all flow is directed towards, and which absorbs it. All other vertices must satisfy a conservation of flow condition: the amount of flow coming into them must be equal to the amount of flow leaving them. The flow travels along edges, and must obey their capacity constraints.

Definition 1 *A* flow network *is defined by a directed graph $G = (V, E)$, a non-negative integer valued capacity c_e for each edge $e \in E$, and distinguished* source *and* sink *nodes $s, t \in V$. Without loss of generality we assume that s has in-degree 0, and t has out-degree 0.*

Definition 2 *Given a flow network, an $s \to t$ flow is a function $f : E \to \mathbb{R}_{\geq 0}$ mapping edges to non-negative real numbers. The value $f(e)$ is the amount of "flow" being carried on edge e. A flow must obey two sets of constraints:*

1. *Capacity: For each edge $e \in E$, $0 \leq f(e) \leq c_e$.*
2. *Flow Conservation: For every $v \in V \setminus \{s, t\}$:*

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

Here "e into v" denotes the set of directed edges $\{(u, v)\}_{u \in V}$ entering v, and "e out of v" denotes the set of directed edges $\{(v, u)\}_{u \in V}$ leaving v.

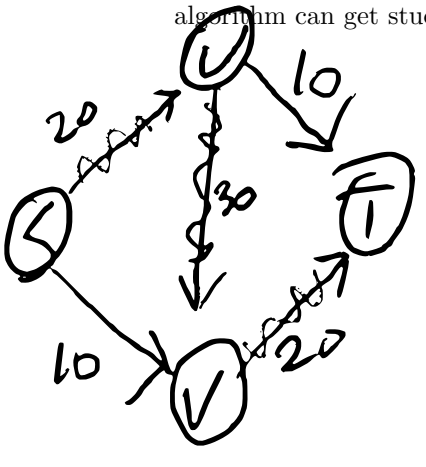
The value *of a flow is the amount of flow leaving the source:*

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

.

The idea here is that flow is generated at (and *only* at) s, and is absorbed at (and *only* at) t. It must be conserved everywhere else, and must obey all edge capacities. Given a flow network, the object of the maximum flow problem is to find the flow of maximum value $v(f)$.

The problem seems to resist the design paradigms we have seen so far. For example, we might consider a "greedy" algorithm that looks at the current flow, finds the path from s to t that has the largest remaining slack (minimum difference between capacity and flow along any edge in the path), and increases the flow along that path to eliminate the slack. But as we see in the following example, this algorithm can get stuck:



Instead, we'd like to be able to make incremental changes to a flow that preserve the flow constraints, but allow us to "undo" decisions we have made earlier. So, we can *remove* flow coming into a vertex so long as we replace it with an equal amount of other incoming flow, or subtract it from the outgoing flow. To make this procedure easy, we define the residual graph:

Definition 3 *Given a flow network G and a flow f, the residual graph G_f is a flow network defined on the same vertex set as follows:*

- *For each edge $e = (u, v)$ in G with $f(e) < c_e$ there is an edge $e = (u, v)$ in G_f with capacity $c_e - f(e)$ (i.e. with capacity equal to the* unused *capacity in G. These are forward edges.*
- *For each edge $e = (u, v)$ in G with $f(e) > 0$, there is a* backwards *edge $e' = (v, u)$ in G_f with capacity $f(e)$.*

Observe that G_f has at most twice as many edges as G. The idea is that when we push flow along a path in G_f, when we use forward edges, we are using up remaining unused capacity in G. When we use backwards edges, we are undoing decisions we previously made producing flow in G, removing along each backwards edge flow up to the amount initially flowing in G. The idea behind our algorithm will be to iteratively build a flow f by repeatedly:

1. Constructing the residual graph G_f,
2. Finding a simple (i.e. acyclic) path P in G_f that has capacity to push more flow, and
3. Appropriately updating f by "pushing" flow along P.

We'll refer to capacities in G_f as *residual capacities*. For a path P and flow f, we write bottleneck(P, f) for the minimum residual capacity of any edge in P with respect to flow f — i.e. the maximum amount of flow we can route along flow P in the residual graph G_f. Here is our update subroutine that we will use to "push" flow along a path in G_f:

Algorithm 1 Augment(f, P)

```
Let b = bottleneck(P, f)
for each edge (u, v) ∈ P do
  if e = (u, v) is a forward edge then
    f(e) = f(e) + b.
  else if (u, v) is a backward edge then
    Let e = (v, u)
    f(e) = f(e) − b
  end if
end for
Return f.
```

The Augment() operation takes as input a flow and an *augmenting path* in the residual graph and produces a new flow. Our final algorithm for computing the maximum flow in a graph G (the "Ford-Fulkerson Algorithm") will just repeatedly make calls to our Augment() subroutine as long as it is able:

Algorithm 2 Max-Flow(G)

```
Initialize an empty flow f(e) = 0 for all e in G.
while there is an s → t path in the residual graph G_f do
  Let P be any simple s → t path in G_f.
  Update f =Augment(f, P).
end while
```

The analysis of this algorithm will take us a little while. We'll start with a modest claim: If Max-Flow(G) halts, then it outputs a valid flow.

Claim 4 *Let G be any flow network. If Max-Flow(G) = f, then f is a flow in G.*

Proof We will prove that if f is a flow in G and P is a simple $s \to t$ path in G_f, then f' =Augment(f, P) is also a valid flow. This suffices to prove the claim, because Max-Flow starts with an empty flow (always a valid flow in G), and then simply iterates the Augment operation on simple $s \to t$ paths.

By definition of Augment, f' differs from f only in edges in P, so it suffices to verify the capacity constraints only for these edges. Recall that b = bottleneck(P, f) is an upper bound on the residual capacity of any edge $(u, v) \in P$. If $e = (u, v)$ is a forward edge, then we have that its residual capacity is $c_e - f(e)$. So we have:

$$0 \leq f(e) \leq f'(e) = f(e) + b \leq f(e) + (c_e - f(e)) = c(e).$$

Similarly, if (u, v) is a backwards edge, then for $e = (v, u)$ we have that its residual capacity is $f(e)$, and:

$$c_e \geq f(e) \geq f'(e) = f(e) - b \geq f(e) - f(e) = 0$$

We also need to check the flow conservation constraints at each node visited by P. Since f satisfied the flow conservation conditions at each vertex, so does f'. Suppose both edges in P neighboring v are forward edges (u, v) and (v, w). Then we push b units of additional flow into v along (u, v) and a corresponding b units of flow out of v along (v, w). If they are both backwards edges, the result is symmetric. Now suppose one of the edges is a forward edge and the other is a backwards edge — e.g. consider the case in which we have a forwards edge (u, v) and a backwards edge (w, v). In this case we add b units of flow into v along (u, v) and subtract b units of flow into v along (w, v). Again we come out even. The remaining case is symmetric. ■

Now that we've established that *if* our algorithm halts, it outputs a feasible flow, we want to establish that it in fact halts. Towards this goal we'll prove a simple lemma:

Lemma 5 *At each intermediate stage of Max-Flow, the flow values $f(e)$ and the residual capacities in G_f are integers.*

Proof This is true at the start of the algorithm: the flow values $f(e) = 0$, and the residual capacities are equal to the capacities c_e which we have assumed are integers.

It remains to prove that if this holds for a flow f, then it also holds for the flow f'=Augment(f, P). Since (by assumption) all of the residual capacities in G_f are integers, then the bottleneck value b will also be an integer. The flow values in f' are derived from adding a value in $\{-b, 0, b\}$ to the flow values in f, and so remain integers, which completes the proof. ■

We now observe that the value of the flow strictly increases with each iteration of the algorithm.

Lemma 6 *Let f be any flow in G, and let P be a simple s-t path in the residual graph G_f. Then if f' =Augment(f, P), $v(f') > v(f)$.*

Proof The first edge in P must be a forward edge leaving s (since s has no incoming edges). The flow on this edge is increased by the bottleneck value $b > 0$. No other edge adjacent to s is modified, and so $v(f') - v(f) = b > 0$. ■

Now, lets define a quantity C that we will use to upper bound the run-time of Max-flow. Let $C = \sum_{e \text{ out of } s} c_e$. We know that for any flow f, $v(f) \leq C$, since any flow of greater value would have to violate the capacity constraint of one of the edges leaving s. This lets us conclude:

Theorem 7 *Given a flow network G with integer capacities, Max-flow terminates in at most C iterations.*

Proof We have established that:

1. The value of the flow strictly increases at each iteration, and
2. The value of the flow is always an integer.

Hence, the value of the flow (initially 0) must increase by at least 1 at every iteration. Since the value of the flow cannot exceed C, this process can continue for at most C iterations. ■

We have now established that our algorithm halts and outputs a valid flow; it remains to prove that it is a flow of maximum value. To do this, we will dive deeper into an investigation into the structure of optimal flows, and what turn out to be an important dual object: cuts.

Definition 8 *Given a flow network $G = (V, E, \{c_e\})$, an $s-t$ cut is a partition of V into two parts $V = A \cup B$ such that $s \in A$ and $t \in V$. The value or* capacity *of a cut is the sum of the capacities of the edges crossing between A and B:*

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

Observe that C defined above is just the value of the cut $(\{s\}, V \setminus \{s\})$. This was an upper bound on the value of the optimal flow. It turns out that the capacity of *any* s-t cut is an upper bound on the value of the optimal flow! This is pretty intuitive if you think about it (at some point the flow has to cross any particular s-t cut, and it can't use more than the total capacity crossing the cut), but we're going to prove it.

First some useful notation. Given a subset of vertices $S \subseteq V$ and a flow f, we write $f^{out}(S) = \sum_{e=(u,v):u \in S, v \notin S} f(e)$ and $f^{in}(S) = \sum_{e=(u,v):u \notin S, v \in S} f(e)$. With this notation we can write:

Claim 9 *Let f be any $s-t$ flow and (A, B) be any $s-t$ cut. Then:*

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

Proof Consider $s \in A$. We have by definition that $v(f) = f^{out}(s)$, and since again by assumption s has no incoming edges, we can also write:

$$v(f) = f^{out}(s) - f^{in}(s).$$

Now observe for any other $v \in A$, we have by the flow conservation constraints that $f^{out}(v) - f^{in}(v) = 0$, and so we may add these terms into the sum:

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

Now consider the edges that appear in this sum. There are two four kinds of edges $e = (u, v)$. If $u, v \notin A$, then the edge appears nowhere in the sum (it is not involved in flow entering or leaving any vertex in A). On the other hand, if $u, v \in A$, then the edge appears once as an incoming edge and once as an outgoing edge, and so the flow along it cancels out. Thus the only edges $e = (u, v)$ whose flow appears with positive sign are those such that $u \in A$ and $v \notin A$, and the only edges whose flow appears with negative sign are those such that $u \notin A, v \in A$. Thus we can write:

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

■

The above claim characterizes the value of an arbitrary flow with respect to an arbitrary s-t cut: it is the value of the flow that manages to cross the cut from A to B, minus the value of any flow that happens to slosh back in. An easy corollary of this is that the capacity of *any* cut is an upper bound on the value of a flow:

Theorem 10 *Let f be any $s-t$ flow and let (A, B) be any s-t cut. Then:*

$$v(f) \leq c(A, b).$$

Proof From our above characterization, we have that:

$$v(f) = f^{out}(A) - f^{in}(A) \leq f^{out}(A) = \sum_{e \text{ out of } A} f(e) \leq \sum_{e \text{ out of } A} c_e = c(A, B)$$

■

Our strategy for proving that the Ford-Fulkerson algorithm always returns a max flow f^* is to demonstrate some s-t cut (A^*, B^*) such that $v(f^*) = c(A^*, B^*)$. Since we know that for *every* flow f, $v(f) \leq c(A^*, B^*)$, this serves as a certificate that f^* is optimal. But our ability to carry out this line of argument does more than that — it teaches us that *flows* and *cuts* are duals of each other in a very strong sense. The maximum value flow in a graph is always exactly equal to the minimum capacity cut!

The Ford-Fulkerson algorithm halts and outputs a flow f^* when there no longer exists any $s-t$ path in the residual graph G_{f^*}. Therefore, to prove the optimality of the algorithm, it suffices for us to prove:

Theorem 11 *If f is an s-t flow such that there does not exist any s-t path in the residual graph G_f, then there is an s-t cut (A^*, B^*) in G such that $v(f) = c(A^*, B^*)$. In particular, f is a maximum value flow in G, and (A^*, B^*) is a minimum value cut.*

Proof We've been given the flow f, and our job is to exhibit the cut. To this end, let A^* denote the set of all vertices v such that there is a path from s to v to G_f. We have $s \in A^*$, and by the hypothesis of the theorem, we have $t \notin A^*$. Let $B^* = V \setminus A^*$. This forms an s-t cut.

Now consider any edge $e = (u, v)$ that crosses the cut in the forwards direction: $u \in A^*, v \in B^*$. It must be that $f(e) = c(e)$. If this were not the case, then there would be a forward edge $e = (u, v)$ in the residual graph G_f with capacity $c(e) - f(e) > 0$. By assumption, there is an s-u path in G_f, so if there were a forward edge (u, v) in G_f, then there would be an s-v path as well. But this would contradict the fact that $v \in B^*$, which consists only of edges v such that there is no s-v path in G_f.

Similarly, consider any edge $e' = (u', v')$ that crosses the cut in the backwards direction — $u' \in B^*$ and $v' \in A^*$. It must be that $f(e') = 0$, since otherwise there would be a backwards edge from v' to u' in G_f, contradicting the premise that there is to path from s to u' in G_f. Therefore, from our characterization of flows by cuts, we have:

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

■

A corollary of this is a striking structural result about graphs, often called the Max-Flow/Min-Cut theorem:

Theorem 12 *In every flow network, the maximum value of any s-t flow is equal to the minimum value of any s-t cut.*

Proof The Ford-Fulkerson algorithm finds a flow f^* and a cut (A^*, B^*) such that $v(f^*) = c(A^*, B^*)$. f^* is a maximum value flow since we know that for every flow f, $v(f) \leq c(A^*, B^*)$. Similarly, (A^*, B^*) is a minimum value cut, since if there were some other cut (A', B') with smaller value, we would contradict the bound $v(f^*) \leq c(A', B')$. ■

We've also proved another structural theorem along the way. Since we argued that the Ford-Fulkerson algorithm always returns an integer valued flow given a network with integer valued capacities, we have:

Theorem 13 *In any flow network with integer valued capacities, there always exists a maximum $s-t$ flow with integer flow values $f(e)$.*

Our ability to compute maximum value flows in arbitrary graphs is powerful, and has a number of applications that you will explore on the homework. Here we'll conclude with just one simple application: computing a maximum cardinality matching in an unweighted bipartite graph.

Recall that a bipartite graph $G = (V, E)$ is one such that the vertex set can be partitioned into left and right sides $V = L \cup R$, and E consists only of edges with one endpoint in L and one endpoint in R. A matching M is a subset of edges $M \subseteq E$ such that each vertex in V is adjacent to at most one edge in M. The goal is to find a matching of maximum cardinality; we gave an auction based algorithm for this in Lecture 2.

Here is another solution. Construct a flow network $G' = (V', E')$ as follows. We have that $V' = V \cup \{s, t\}$, and $E \subset E'$ (each with orientation going from L to R). We also add to E' an edge (s, u) for each $u \in L$ and an edge (v, t) for each edge $v \in R$. All edges e have capacity $c_e = 1$. A picture is useful here.

Our algorithm will simply compute a maximum flow in G', and read the matching off of the flow. We'll see how to do this and why this works with a few simple observations:

First, suppose there exists a matching M in G consisting of k edges $M = (e_1, \ldots, e_k)$. Then there exists a flow of value k in G'. For each edge $e_i = (u_i, v_i)$, we set $f(s, u_i) = f(u_i, v_i) = f(v_i, t) = 1$. By construction we have $v(f) = k$, and the capacity constraints are satisfied since $f(e) \leq 1 = c_e$ for each edge. It only remains to verify the flow conservation constraints, which are also satisfied by construction since M is a matching.

We need to argue the other direction as well: if f is a flow of value k, then we can read off from it a matching M of cardinality k. We will let $M = \{e \in E : f(e) > 0\}$. We make two simple observations:

Claim 14 *M contains k edges.*

Proof Our maximum flow algorithm returns an integer valued flow, and so every edge $e \in E$ has flow value $f(e) \in \{0, 1\}$. The flow crossing the cut (L, R) has value exactly equal to k, and so there must be k edges with nonzero flow. ■

Claim 15 *M is a matching.*

Proof Suppose otherwise — without loss of generality that there is a vertex $u \in L$ such that there are two vertices $v_1, v_2 \in R$ with $(u, v_1), (u, v_2) \in M$ (the other case is symmetric). Since the flow was integer valued, this means there were 2 units of flow leaving u. By flow conservation, there must also have been 2 units of flow incoming, but this cannot be, since the capacity of the edge $e = (s, u)$ is $c_e = 1$. ■