Colorado CSCI 5454: Algorithms September 12 and 17, 2019

Lecture 6-7

Lecturer: Bo Waggoner *Scribe: Bo Waggoner*

Max Flow, Min Cut, and Bipartite Matching

1 The Max Flow Problem

The *max s to t flow problem* captures routing large amounts of traffic on a graph along multiple paths from a source vertex s to a sink vertex t. Different edges in the graph have different capacities of how much flow they can handle.

Example: We have an Internet network where each wire can support a certain bandwidth in bits/second. We wish to send as many bits/second from s to t as possible, and we can split the routing of these bits over many different paths.

Input: A graph $G = (V, E)$ with a capacity function[1] $c : E \to \mathbb{R}_{\geq 0}$. Also, two vertices s (the source) and t (the sink).

If we have an edge $e = (u, v)$, then we may write either $c(e)$ or $c(u, v)$ for the capacity of that edge. If there is no edge from u to v or if $u = v$, we define $c(u, v) = 0$. So c is well-defined on all pairs of vertices.

Output: the amount of the **maximum flow** from s to t. A *flow* is a function $f : E \to \mathbb{R}_{\geq 0}$ such that:

- (capacity constraint) $f(e) \leq c(e)$ for all e.
- (flow constraint) For each $v \notin \{s, t\}$, we have $\sum_{u \in V} f(u, v) = \sum_{w \in V} f(v, w)$.

Like with the capacity function, if $u, v \in V$ and (u, v) is not an edge in the graph, we define $f(u, v) = 0$.

The *amount* of a flow is the total flow going into the sink, which is $\sum_{v \in V} f(v, t)$. A *maximum flow* is a flow with the largest possible amount.

The capacity constraint says we send no more flow along a particular edge than it can handle. The flow constraint says that for any intermediate vertex, the total amount flowing in has to equal the total amount flowing out.

Internet routing example: The capacity constraint says that we can't send more bits/second over a wire than its capacity allows. The flow constraint says that if a certain number of bits/second flow into a router, the same total amount has to flow out.

Assumptions and variants. There are many variants on the problem that might seem challenging, but can actually be reduced to a simple version.

- Suppose there were multiple sources and we can send as much flow as we want out of each. Then we can create a copy of the graph and add a single "supersource" vertex with infinite-capacity edges to all the original sources, which are now ordinary intermediate nodes. Now we have a problem with a single source. But if we solve it, the answer will be identical to the solution to the original multi-source problem.
- Similarly, we can accomodate multiple sinks as well (where the goal is to send the maximum total flow into all sinks combined). We create a new "supersink" vertex with infinite-capacity edges from all the original sinks, which are now ordinary nodes.

[1] Note that c can be represented in the input as edge weights.

- If there are edges into s and edges out of t, these cannot change the optimal solution. So we can remove them without changing the answer.
- Suppose there are vertices not on any path from s to t. These can never be used to route flow, so we can remove without changing the answer.
- Suppose there are "antiparallel" edges (i.e. an edge from u to v and from v to u, i.e. $c(u,v), c(u,v) > 0$). We can create a version of the graph without any antiparallel edges and the same max-flow solution. For each edge $e = (u,v)$ and antiparallel companion (v,u), we create a new artificial intermediate node v_e. Now we can create directed edges (u, v_e) and (v_e, v) with the same capacity as the original edge e. However, the capacities backward along these two edges are zero. Meanwhile, we set the capacity along e to zero. *(If this is hard to follow, draw a diagram with u, v and the two antiparallel edges; then add v_e and the new edges.)*

So **we can assume our problem has only one source, one sink, all vertices are on some path $s \to t$, and there are no antiparallel edges**. If we solve this problem, then by the above logic, all these extensions are easy.

2 Solution - idea

A natural idea is to start with zero flow: $f(e) = 0$ everywhere. Then, find a path from s to t where we can add a little bit of flow everywhere along the path. For example, maybe the smallest capacity on any edge in this path is 3. Then we can route 3 units of flow along each of the edges of the path. Now, we still have a valid flow (show that both constraint are still satisfied!), and the amount of flow is larger, i.e. 3. So if we find another path where there is room to add, and keep repeating this, hopefully we will stop at the maximum possible flow.

This idea is pretty good, but it won't actually quite work. How can we derive the actual correct version? Answer: math! We need to know when the flow we've found is optimal, as well as a method for "augmenting" the current flow correctly. So the next step is to prove some facts about flows on graphs.

3 Min Cuts

The idea is to formalize *bottlenecks* in the graph: places that all of the flow has to squeeze through simultaneously to get to the destination. For example, if all Internet traffic from North America to Europe has to go through the Atlantic undersea cable, then the max flow from any city in Canada to any city in Sweden will be at most the capacity of that cable (no matter how big the wires are between Toronto and Montreal or so on).

Definitions. A *cut* is a partition of the vertices of the graph into two sets, S and T. It is called an s-t cut if $s \in S$ and $t \in T$.

Define the *value* of the cut to be $K(S,T) := \sum_{u \in S} \sum_{v \in T} c(u,v)$, i.e. the sum of all capacities of all edges across the cut from S to T.

The *minimum $s-t$ cut* is the cut S, T minimizing $K(S,T)$. In other words, you are allowed to split the vertices in any way as long as $s \in S$ and $t \in T$, and your goal is to find the split with the smallest amount of total capacity crossing from S over to T.

Formally, the min $s-t$ cut problem gives as input a directed, weighted graph $G = (V, E)$ with edge weights $c : E \to \mathbb{R}_{\geq 0}$, along with vertices s, t. The output is the value of the min $s-t$ cut.

3.1 Max flow - min cut theorem

Given a legal flow $f(u,v)$ and a cut (S,T), write the *net flow across the cut*

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

Lemma 1. *For any $s-t$ cut S,T, we have $F(S,T) \leq K(S,T)$.*

Proof.

$$\begin{aligned} F(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - f(v,u) \\ &\leq \sum_{u\in S}\sum_{v\in T} f(u,v) && \text{(because each } f(v,u)\geq 0\text{)} \\ &\leq \sum_{u\in S}\sum_{v\in T} c(u,v) && \text{(capacity constraint)} \\ &= K(S,T). \end{aligned}$$

Let us verbally retrace the lines of this proof. First, the net flow across the cut is at most the total flow going from S to T, i.e. we stop subtracting any flow from T back to S. Second, the flow on each edge from S to T has to be at most the capacity of the edge, because it's a valid flow. Then we arrive at the definition of the value of the cut. □

Okay, so far so good. But what is the net flow across some particular cut?

Lemma 2. *For any flow f, the net flow across any $s-t$ cut is the same. Furthermore, it equals the amount of the flow (recall the amount is $\sum_v f(v,t)$).*

To see the proof idea, suppose we begin with $S = \{s\}$, the cut with just s on one side and every other vertex on the other. Now move a single vertex v over to S. How does the net flow change? We no longer have the edge from s to v across the cut, so it decreases by that amount. On the other hand, we have all the edges from v out to the rest of the graph, so it increases by that amount. Because we know the net flow into v equals the net flow out, it turns out that we haven't changed the net flow across the cut! Repeating this idea vertex by vertex will prove the claim.

Proof. Consider any cut S,T where T contains a vertex $v \neq t$. Now consider the cut $S' = S \cup \{v\}, T' = T \setminus \{v\}$. In other words, we move v over to S.

What is the difference between $F(S,T)$ and $F(S',T')$? All edges that do not include v have not changed, so we only need to consider edges involving v. In $F(S',T')$, we only have edges from v to T', so the net flow on these edges is $\sum_{w\in T'} f(v,w) - f(w,v)$. Meanwhile, in $F(S,T)$, we only have edges from S to v, so the net flow on these edges is $\sum_{u\in S} f(u,v) - f(v,u)$. So

$$\begin{aligned} F(S',T') - F(S,T) &= \left[\sum_{w\in T'} f(v,w) - f(w,v)\right] - \left[\sum_{u\in S} f(u,v) - f(v,u)\right] \\ &= \sum_{v'\in V} f(v,v') - f(v',v) \\ &= 0 \end{aligned}$$

by the flow constraints, because f is a valid flow.

This proves that $F(S,T)$ stays the same when we move one vertex over from T to S. Now if we start from $S = \{s\}$ and move vertices into S one at a time, we can arrive at any S,T cut and the net flow has not changed. We can even get to $T = \{v\}$. But in that case, $F(S,T) = \sum_v f(v,t)$ which is the amount of the flow f. This proves the lemma. □

Lemmas 1 and 2 imply:

Lemma 3. *The amount of any $s-t$ flow is less than or equal to the value of any $s-t$ cut.*

Proof. By Lemma 2, if we have an $s-t$ flow, it sends the same net flow (its amount) across any cut. By Lemma 1, the net flow is smaller than the value of the cut. □

Now let's remember the idea we had for an algorithm: gradually increase the flow along certain paths. This helps us know when to stop: if we have an amount of flow equal to *any* cut, then we know we can't do any better! The proof that this is possible to achieve will give us an algorithm.

Theorem 1 (Max-flow min-cut theorem)**.** *In any graph satisfying our assumptions, the max $s-t$ flow is equal to the min $s-t$ cut.*

Proof. We show that, if a flow amount is smaller than the min cut, then it can be improved. This will prove that the max flow must be at least the min cut. On the other hand, by Lemma 3, the max flow is at most the min cut, so they must be equal.

Given a flow f with amount smaller than the min cut, start with $S = \{s\}$ and $T = V \setminus \{s\}$. Now we have some cut S, T with $F(S,T) < K(S,T)$. This says

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

So in particular there is some pair $u \in S, v \in T$ with $f(u,v) - f(v,u) < c(u,v)$. We add v to S. Now again we have a cut S,T with $F(S,T) < K(S,T)$ so we repeat the process until we finally add t. The key point is that we will always be able to make progress because the flow is less than every cut.

If we take the sequence of vertices added to S, possibly after discarding some, we obtain a simple path $u_1, \dots, u_k$ where $s = u_1$ and $t = u_k$. For each step in the path, we have $f(u,v) - f(v,u) < c(u,v)$. So we can increase the net flow across each edge in this path by some small constant amount without violating the flow constraints: If $f(v,u) > 0$ we can decrease it, and if $f(v,u) = 0$ while $f(u,v) > 0$, we can increase $f(u,v)$. In either case, we can increase the amount of the flow. (Check that the capacity and flow constraints are still satisfied!)

So we have shown that any flow with amount smaller than the min cut is not a max flow, proving the theorem. $\square$

4 Residual Graph and Ford Fulkerson framework

It's time to create an algorithm! We leverage the ideas in the proof of the max-flow min-cut theorem.

Given a flow f, define the *residual network* $G_f = (V, E')$ where the vertices are the same and for each original edge $(u,v) \in E$, we have both edges (u,v) and (v,u) in E'. (This is why we required no antiparallel edges in the original network.)

The residual network has an associated *residual capacity* $c_f : E' \to \mathbb{R}_{\geq 0}$. For each edge $e = (u,v)$ in the original graph, we set

$$c_f(u,v) = c(u,v) - f(u,v)$$
$$c_f(v,u) = f(u,v).$$

In other words, along the edges (u,v) of the original graph, we set $c_f(u,v)$ equal to how much room we have to increase the flow before hitting capacity. But we also record how much room we have to *decrease* the flow, the weight in the opposite direction, $c_f(v,u)$.

Again, $c_f(e) = 0$ is treated identically to the edge not existing in E', so if $c(u,v) = f(u,v)$, then we say there is no edge (u,v) in E'. And if $f(u,v) = 0$, then we say there is no edge (v,u).

Now we can give the *Ford-Fulkerson framework*: Begin with flow f zero everywhere, and repeat:

- Build the residual graph G_f.
- Find a simple path from s to t in the residual graph. If no such path can be found, stop the algorithm and return f.
- Let $z =$ the smallest weight $c_f(u,v)$ of any edge along that path.
- Augment the flow f by adding z along each edge of the path. That is, for each edge (u,v) in the path: If $(u,v) \in E$, then increase $f(u,v)$ by z. If not, then we must have that $(v,u) \in E$ and there is currently some amount of flow from v to u. So we decrease the flow $f(v,u)$ by z.
- Repeat.

Correctness: If the algorithm terminates, we claim it is correct. We always have a valid flow: For capacity constraints, we keep $f(u,v) \leq c(u,v)$ for every edge $(u,v) \in E$ and $f(u,v) = 0$ if $(u,v) \notin E$; note that $c_f(u,v) = c(u,v) - f(u,v)$ if $(u,v) \in E$. For flow constraints, initially we have flow zero everywhere and they are satisfied. Thereafter, along any path from s to t, consider an intermediate vertex v with edges of the path (u,v) and (v,w). If $(u,v) \in E$, we increase the flow from v into u by z. If $(v,u) \in E$, we *decrease* the flow *out* of u into v by z. In either case, the *net* flow into v increases by z; and similarly along the edge between u and w, the net flow out of v increases by z, so the flow constraint is always satisfied.

Now suppose the algorithm terminates, i.e. at some point we cannot find a simple path in the flow graph. Then there must be a cut S,T where any edge from some $u \in S$ to some $v \in T$ has $f(u,v) - f(v,u) = c(u,v)$. (Otherwise, a greedy algorithm could always make progress toward finding a simple path from s to t, just as in the proof of the max-flow min-cut theorem.) In other words, $F(S,T) = K(S,T)$, so we have a flow that equals a cut, so it is a max flow.

Halting and running time: The proof that the algorithm halts, and its running time, depends on how you find the augmenting path in the residual graph in each iteration. One approach leads to the Edmonds-Karp algorithm.

4.1 Edmonds-Karp

The *Edmonds-Karp* algorithm is a case of the Ford-Fulkerson framework. It finds an augmenting path via breadth-first search in the residual graph G_f from s to t.

- Treat G_f as a directed, unweighted graph (but only include edges with nonzero weight).
- Use breadth-first search to find a shortest path p (one with the fewest edges) from s to t in G_f.
- Let z be the smallest weight of any edge in p.
- Augment the flow f by adding z to each edge in p.

Correctness has already been argued for all algorithms in the Ford-Fulkerson framework.

Halting and running time: This proof is a bit involved, so we prove it in steps.

Given a residual graph G_f and an augmenting path p chosen by the algorithm, call an edge in the path *critical* if it achieves the minimum value of $c_f(u,v)$ along that path. Note that, in each iteration, the residual graph G_f changes by removing each critical edge (u,v) of the augmenting path, because their residual flow becomes zero after the augmentation; and it also changes by adding the reverse edge (v,u), because the augmentation creates residual flow in the reverse direction.

Lemma 4. *In each iteration of the Edmonds-Karp algorithm, any vertex's distance from s (i.e. number of hops) in the residual graph cannot decrease.*

Proof. Let (u,v) be a critical edge in an augmenting path. This is a shortest path from s, so in general u is, say, distance k from s while v is distance $k+1$. After the augmentation, (u,v) is removed from the graph, which can only increase the distance of v (and any other vertex). The edge (v,u) is added, but this cannot shorten any paths in the graph because the shortest path to u has distance k, whereas going through v would be strictly farther. □

Lemma 5. *In the Edmonds-Karp algorithm, each edge that may be present in the residual graph can be critical in at most $|V|/2$ iterations.*

Proof. Consider a potential edge $e = (u,v)$ and suppose it is critical in some iteration of the algorithm. Let the shortest-path distance from s to u at this time be k, and the distance to v is therefore $k+1$. After the augmentation, e is removed from the residual graph.

Now, e can only be added back to the residual graph if the reverse edge (v,u) becomes critical. But in this case, it must be that a shortest path from s to u goes through v, so u has distance at least $k+2$.

Repeating this, we see that each time (u,v) becomes critical, the distance of u from s has increased by 2, and this can occur at most $n/2$ times because the distance never exceeds n. □

Theorem 2. *The Edmonds-Karp algorithm on graph $G = (V, E)$ has running time $O(|V||E|^2)$.*

Proof. For each edge (u, v) of the original graph, we may have an edge (u, v) or (v, u) in the residual graph. By Lemma 5, each of these $2E$ edges is critical at most $|V|/2$ times, so there can be no more than $|E| \cdot |V|$ iterations. Each iteration can be accomplished via a call to breadth-first search on a graph with $|V|$ vertices and at most $2|E|$ edges, taking $O(|V| + |E|)$ time. (One may check that constructing the residual graph also takes linear time; one can also construct it "lazily" just-in-time.) Now, $|E| \geq |V| - 1$ (because we assume every vertex is on a path from s to t), so each iteration takes $O(|E|)$ time, giving a total of $O\left(|V| \cdot |E|^2\right)$ time. □

5 Finding a Min Cut

The max-flow min-cut theorem essentially tells us how to find a min cut S, T:

- Find a max flow f
- Construct the residual graph G_f
- Let S be the set of vertices reachable from s in the residual graph; let T be the remainder.

Correctness: By the max-flow min-cut theorem, we need to find a cut S, T where $F(S, T) = K(S, T)$. Start with $S = \{s\}$. Any vertex v we can reach from S in the residual graph must have $f(u, v) - f(v, u) < c(u, v)$ for some $u \in S$, so $F(S, T) < K(S, T)$ so it's not a min cut. So add v to S and repeat. At some point we cannot reach any more vertices meaning $f(u, v) = c(u, v)$ for all edges across the cut.

Running time: Once we have the flow f, constructing the residual graph and executing BFS or DFS on it are linear-time operations, so the big-O time complexity is the same as of max cut.

6 Bipartite Matching

6.1 Definitions

A graph G is *bipartite* if we can divide the vertices into two sets, call them U and V, such every edge has exactly one endpoint in U and one endpoint in V. We will often write such graphs $G = (U, V, E)$.

Examples of bipartite graphs:

- A *star graph* has a central vertex u and a bunch of outer vertices $v_1, \ldots, v_{n-1}$. There is an edge between u and v_i for each i, and these are all the edges.
- A graph consisting of a pair (u_1, v_1) and the edge between them; a separate pair (u_2, v_2) and the edge between them; and so on.
- Any tree is bipartite (can you see why?).

Examples of non-bipartite graphs:

- A triangle graph (three vertices, with an edge between each pair).
- Any complete graph (all possible edges present) on $n \geq 3$ vertices, because it contains a triangle.
- Any graph that contains an odd-length cycle.

In any graph, a *matching* is a set of edges M that have no vertices in common. A vertex is *matched* if it is the endpoint of one of the edges in the matching. Otherwise, it is *unmatched*. A matching is *perfect* if every vertex in the graph is matched.

Examples:

- The empty set is a matching (nobody is matched to anybody).

- A set containing one edge is a matching.
- In a star graph, the largest possible matching has size just one. (Do you see why?)

Motivating example: We have a set of people U and jobs V. If person u is qualified to do job v, we have an edge (u, v). A matching is an assignment of each person to at most one job and vice versa.

6.2 Maximum matchings

In this problem, the input is an unweighted, undirected bipartite graph $G = (U, V, E)$ and the output is a matching M of largest possible size.

To solve this problem, we will simply reduce to the max flow problem. (Note there are more efficient, specialized combinatorial algorithms, but the "augmenting paths" ideas they use are quite similar to max flow.)

Algorithm. Given the bipartite graph, we construct a weighted, directed graph G', which we call the *flow version* of G. In G', we have all vertices in U and V, and for each $(u, v) \in E$, we create a directed edge (u, v) with capacity $c(u, v) = \infty$. (If we prefer, we can just make it a very large integer like $2n$.) We then create a "source vertex" s and create an edge (s, u) for each $u \in U$ with capacaity $c(s, u) = 1$. Finally, we create a "sink vertex" t with an edge (v, t) for each $v \in V$ with capacity $c(v, t) = 1$.

Then, we simply run a max-flow algorithm on G' with capacities c and source s, sink t. We return the value of the maximum flow as the size of the maximum matching, and the edges of the form (u, v) used in that flow as the edges of the matching.

Correctness - wait a minute! Counterexample: one can construct a non-integral max flow. For example, $U = \{a, b\}$ and $V = \{c, d\}$ with all edges $E = \{(a, c), (a, d), (b, c), (b, d)\}$. We can have a flow $f(e) = 0.5$ along each of these edges $e \in E$, which is a max flow, but doesn't correspond to any matching.

Luckily, we can show that in these cases, our algorithms find an *integral* max flow.

Theorem 3 (Integrality theorem)**.** *In a max-flow instance where each capacity $c(u, v)$ is an integer, any algorithm in the Ford-Fulkerson framework assigns an integral amount of flow $f(u, v)$ to every edge. In particular, the max flow amount is an integer.*

Proof. We simply need to argue that each $f(u, v)$ and each residual capacity $c_f(u, v)$ is an integer throughout the entire algorithm. At the beginning, $f(u, v) = 0$ and $c_f(u, v) = c(u, v)$. Now inductively: At each iteration, the amount of augmenting flow that is added to f along a path equals $c_f(u, v) - f(u, v)$, for some critical edge (u, v). By assumption, each of these are integers, so the augmentation amount is an integer. So f remains integral everywhere; and the change to each $c_f(u, v)$ along the path is by this same integer, so it also remains integral. □

Correctness - revisited.

Theorem 4. *The max flow (and min cut) of the flow graph G' is equal to the size of the maximum matching in G. Furthermore, the edges $(u, v) \in E$ assigned positive flow by a Ford-Fulkerson algorithm comprise a maximum matching.*

Proof. By the integrality theorem, the output of the max flow algorithm is integral. The flow constraint says that each $u \in U$ has at most one outgoing edge with flow 1 (because it only has incoming flow at most 1 from s), and each $v \in V$ has at most one incoming edge with flow 1. So each vertex has at most one neighboring edge with positive flow, so this gives a valid matching. On the other hand, any valid matching satisfies the flow constraints by the same reasoning. So the max flow gives the maximum matching. □

Efficiency. Constructing the input to the max flow problem takes linear time, so the big-O complexity is equal to the complexity of solving max flow.

6.3 Hall's Theorem

An important result in mathematics is Hall's "marriage" theorem on when a perfect matching exists in a bipartite graph. (Recall that a matching is *perfect* if all vertices participate.) It says this can occur if and only if any subset of c vertices on each side have at least c choices total on the other side.

Given a set of vertices X, define $N(X)$ to be the set of *neighbors* of X, i.e. vertices that share an edge with some vertex in X.

Theorem 5 (Hall's Theorem)**.** *A bipartite graph $G = (U, V, E)$ with $|U| = |V|$ has a perfect matching if and only if, for every $X \subseteq U$, $|N(X)| \geq |X|$.*

In other words, each left vertex must have at least one neighbor on the right; each set of two left vertices must have at least two neighbors on the right total; etc.

Proof. The reverse direction is easy: if there is a perfect matching, then every set $X \subseteq U$ is matched to $|X|$ different vertices in V, so $|N(X)| \geq |X|$.

For the forward direction, consider the flow version G' of G. We prove that the min cut S, T has value at least $|U|$. This implies the max flow is at least $|U|$, which implies by our previous analysis that G has a perfect matching.

So given a min cut S, T, let k be the number of edges from s that cross the cut. This adds k to the value of the cut.

Of the remaining $n - k$ vertices in U, by assumption, they have at least $n - k$ neighbors in V. Furthermore, because this is a min cut, all of these neighbors are in S (otherwise an ∞ edge would be included). Their edges to t add at least $n - k$ to the value of the cut. So the cut has total value at least n. □