Lecture 11

In which we prove that the Edmonds-Karp algorithm for maximum flow is a strongly polynomial time algorithm, and we begin to talk about the push-relabel approach.

1 Flow Decomposition

In the last lecture, we proved that the Ford-Fulkerson algorithm runs in time

$$O(|E|^2 \log |E| \log opt)$$

if the capacities are integers and if we pick, at each step, the fattest path from s to t in the residual network. In the analysis, we skipped the proof of the following result.

Theorem 1 *If $(G = (V, E), s, t, c)$ is a network in which the cost of the maximum flow is opt, then there is a path from s to t in which every edge has capacity at least $opt/|E|$.*

We derive the theorem from the following result.

Lemma 2 (Flow Decomposition) *Let $(G = (V, E), s, t, c)$ be a network, and f be a flow in the network. Then there is a collection of feasible flows $f_1, \ldots, f_k$ and a collection of $s \to t$ paths $p_1, \ldots, p_k$ such that:*

1. *$k \leq |E|$;*
2. *the cost of f is equal to the sum of the costs of the flows f_i*
3. *the flow f_i sends positive flow only on the edges of p_i.*

Now we show how to prove Theorem 1 assuming that Lemma 2 is true.

We apply Lemma 2 to the maximum flow f of cost opt, and we find flows $f_1, \ldots, f_k$ and paths $p_1, \ldots, p_k$ as in the Lemma. From the first two properties, we get that there is an i_0 such that the cost of the flow f_{i_0} is at least $opt/|E|$. From the third

property, we have that the $\geq opt/|E|$ units of flow of f_{i_0} are carried using only the path p_{i_0}, and so every edge of p_{i_0} must have capacity at least $opt/|E|$.

It remains to prove the Lemma.

PROOF: (Of Lemma 2) Now we see how to construct flows with the above three properties. We do so via the following procedure:

- $i := 1$
- $r := f$
- while $cost(r) > 0$
  - find a path from s to t using only edges (u, v) such that $r(u, v) > 0$, and call it p_i
  - let $f_{\min}$ be the minimum of $r(u, v)$ among the edges $(u, v) \in p_i$
  - let $f_i(u, v) := f_{\min}$ for each $(u, v) \in p_i$ and $f_i(u, v) := 0$ for the other edges
  - let $r(u, v) := r(u, v) - f_i(u, v)$ for each (u, v)
  - $i := i + 1$

The "residual" flow r is initialized to be equal to f, and so its cost is the same as the cost of f. At every step i, if the cost of r is still positive, we find a path p_i from s to t entirely made of edges with positive flow.

(Note that such a path must exist, because, if not, call A the set of nodes reachable from s using edges (u, v) that have $r(u, v) > 0$; then A contains s and it does not contain t, and so it is a cut and the net flow out of A is equal to cost of r; but there is no positive net flow out of A, because all the edges from vertices of A to vertices not in A must have $r(u, v) = 0$; this means that the cost of r must also be zero, which is a contradiction.)

We define the flow f_i by sending along p_i the smallest of the amounts of flow sent by r along the edges of p_i. Note that f_i is feasible, because for every edge we have $f_i(u, v) \leq r(u, v)$ and, by construction, we also have $r(u, v) \leq f(u, v)$, and f was a feasible flow and so $f(u, v) \leq c(u, v)$. We then decrease $r(u, v)$ by $f_i(u, v)$ on each edge. This is still a feasible flow, because we leave a non-negative flow on each edge and we can verify that we also maintain the conservation constraints. After the update, the cost of r decreases precisely by the same amount as the cost of f_i, so we maintain the invariant that, after i steps, we have

$$cost(f) = cost(r) + cost(f_1) + \cdots + cost(f_i)$$

It remains to observe that, after the update of r, at least one of the edges that had positive $r(u, v) > 0$ has now $r(u, v) = 0$. (This happens to the edge, or edges, that

carry the minimum amount of flow along p_i.) This means that, after i steps, the number of edges (u, v) such that $r(u, v) > 0$ is at most $|E| - i$ and that, in particular, the algorithm halts within at most $|E|$ iterations.

Call k the number of iterations after which the algorithm halts. When the algorithm halts, $cost(r) = 0$, and so we have

$$cost(f) = cost(f_1) + \cdots cost(f_k)$$

and so the flows and paths found by the algorithm satisfy all the requirements stated at the beginning. □

The running time $O(|E|^2 \log |E| \log opt)$ is not terrible, especially considering that it is a worst-case estimate and that often one has considerably faster convergence in practice. There is, however, an undesirable feature in our analysis: the running time depends on the actual values of the numbers that we get as input. An algorithm for a numerical problem is called *strongly polynomial* if, assuming unit-time operations on numerical quantities, the running time is at most a polynomial in the number of numerical quantities that we are given as input. In particular, a maximum flow algorithm is strongly polynomial if it runs in time polynomial in the number of edges in the network.

Today we describe the Edmonds-Karp algorithm, which is a simple variant of the Ford-Fulkerson algorithm (the variant is that, in each iteration, the $s \to t$ path in the residual network is found using a BFS). The Edmonds-Karp algorithm runs in strongly polynomial time $O(|V| \cdot |E|^2)$ in a simple implementation, and the worst-case running time can be improved to $O(|V|^2 \cdot |E|)$ with some adjustments.

We then begin to talk about an approach to solving the maximum flow problem which is rather different from the Fulkerson-Ford approach, and which is based on the "push-relabel" method. A simple implementation of the push-relabel method has running time $O(|V|^2 \cdot |E|)$, and a more sophisticated implementation has worst-case running time $O(|V|^3)$. We will only present the simpler algorithm.

2 The Edmonds-Karp Algorithm

The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson algorithm in which, at every step, we look for an $s \to t$ path in the residual network using BFS. This means that, if there are several possible such paths, we pick one with a minimal number of edges.

From now on, when we refer to a "shortest path" in a network, we mean a path that uses the fewest edges, and the "length" of a path is the number of edges. The "distance" of a vertex from s is the length of the shortest path from s to the vertex.

BFS can be implemented in $O(|V + E|) = O(|E|)$ time, and so to complete our analysis of the algorithm we need to find an upper bound to the number of possible iterations.

The following theorem says that, through the various iterations, the length of the shortest path from s to t in the residual network can only increase, and it does increase at the rate of at least one extra edge in the shortest path for each $|E|$ iterations.

Theorem 3 *If, at a certain iteration, the length of a shortest path from s to t in the residual network is ℓ, then at every subsequent iteration it is $\geq \ell$. Furthermore, after at most $|E|$ iterations, the distance becomes $\geq \ell + 1$.*

Clearly, as long as there is a path from s to t, the distance from s to t is at most $|V|-1$, and so Theorem 3 tells us that, after at most $|E| \cdot (|V| - 1)$ iterations, s and t must become disconnected in the residual network, at which point the algorithm terminates. Each iteration takes time $O(|E|)$, and so the total running time is $O(|V| \cdot |E|^2)$.

Let us now prove Theorem 3.

PROOF: Suppose that, after some number T of iterations, we have the residual network $R = (V, E'), s, t, c')$ and that, in the residual network, the length of the shortest path from s to t is ℓ. Construct a BFS tree starting at s, and call V_1, V_2, ..., V_k, ..., the vertices in the first, second, k-th, ..., layer of the tree, that is, the vertices whose distance from s is 1, 2, ..., and so on. Note that every edge (u, v) in the network is such that if $u \in V_i$ and $v \in V_j$ then $j \leq i + 1$, that is, nodes can go from higher-numbered layer to lower numbered layer, or stay in the same layer, or advance by at most one layer.

Let us call an edge (u, v) a *forward* edge if, for some i, $u \in V_i$ and $v \in V_{i+1}$. Then a shortest path from s to t must use a forward edge at each step and, equivalently, a path that uses a non-forward edge at some point cannot be a shortest path from s to t.

What happens at the next iteration $T + 1$? We pick one of the length-ℓ paths p from s to t and we push flow through it. In the next residual network, at least one of the edges in p disappears, because it has been saturated, and for each edge of p we see edges going in the opposite direction. Now it is still true that for every edge (u, v) of the residual network at the next step $T+1$, if $u \in V_i$ and $v \in V_j$, then $j \leq i+1$ (where $V_1, \ldots$ are the layers of the BFS tree of the network at iteration T), because all the edges that we have added actually go from higher-numbered layers to lower-numbered ones. This means that, at iteration $T + 1$ the distance of t from s is still at least ℓ, because $t \in V_\ell$ and, at every step on a path, we can advance at most by one layer.

(Note: we have proved that if the distance of t from s is ℓ at one iteration, then it is at least ℓ at the next iteration. By induction, this is enough to say that if will always be at least ℓ in all subsequent iterations.)

Furthermore, if there is a length-ℓ path from s to t in the residual network at iteration $T+1$, then the path must be using only edges which were already present in the residual network at iteration T and which were "forward edges" at iteration T. This also means that, in all the subsequent iterations in which the distance from s to t remains ℓ, it is so because there is a length-ℓ path made entirely of edges that were forward edges at iteration T. At each iteration, however, at least one of those edges is saturated and is absent from the residual network in subsequent steps, and so there can be at most $|E|$ iterations during which the distance from s to t stays ℓ. $\square$

3 The Push-Relabel Approach

All maximum flow algorithms are based on the maximum flow – minimum cut theorem, which says that if there is no $s \to t$ path in the residual network then the flow is optimal. Our goal is thus to "simply" find a flow such that t is unreachable from s in the residual network.

In algorithms based on the Ford-Fulkerson approach, we keep at every step a feasible flow, and we stop when we reach a step in which there is no $s \to t$ path in the residual network.

In algorithms based on the *push-relabel* method, we take a somewhat complementary approach: at every step we have an assignment of flows to edges which is not a feasible flow (it violates the conservation constraints), which is called a *preflow*, but for which we can still define the notion of a residual network. The algorithm maintains the condition that, at every step, t is not reachable from s in the residual network. The algorithm stops when the preflow becomes a feasible flow.

The basic outline of the algorithm is the following: we begin by sending as much flow out of s as allowed by the capacities of the edges coming out of s, without worrying whether all that flow can actually reach t. Then, at each iteration, we consider nodes that have more incoming flow than outgoing flow (initially, the neighbors of s), and we route the excess flow to their neighbors, and so on. The idea is that if we attempted to send too much flow out of s in the first step, then the excess will eventually come back to s, while t will receive the maximum possible flow. To make such an idea work, we need to make sure that we do not keep sending the flow in circles, and that there is a sensible measure of "progress" that we can use to bound the running time of the algorithm.

A main idea in the algorithm is to associate to each vertex v a *height* $h[v]$, with the intuition that the flow wants to go downhill, and we will take the action of sending extra flow from a vertex u to a vertex v only if $h[u] > h[v]$. This will help us avoid pushing flow around in circles, and it will help us define a measure of "progress" to bound the running time.

Here is the outline of the algorithm. Given an assignment of flows $f(u, v)$ to each edge (u, v), and a vertex v, the *excess flow* at v is the quantity

$$e_f(v) := \sum_u f(u, v) - \sum_w f(v, w)$$

that is, the difference between the flow getting into v and the flow getting out of v. If f is a feasible flow, then the excess flow is always zero, except at s and t.

- Input: network $(G = (V, E), s, t, c)$
- $h[s] := |V|$
- for each $v \in V - \{s\}$ do $h[v] := 0$
- for each $(s, v) \in E$ do $f(s, v) := c(s, v)$
- while f is not a feasible flow
  - let $c'(u, v) = c(u, v) + f(u, v) - f(v, u)$ be the capacities of the residual network
  - if there is a vertex $v \in V - \{s, t\}$ and a vertex $w \in V$ such that $e_f(v) > 0$, $h(v) > h(w)$, and $c'(v, w) > 0$ then
    - push $\min\{c'(v, w), e_f(v)\}$ units of flow on the edge (v, w)
  - else, let v be a vertex such that $e_f(v) > 0$, and set $h[v] := h[v] + 1$
- output f

As we said, the algorithm begins by pushing as much flow to the neighbors of s as allowed by the capacities of the edges coming out of s. This means that we get some vertices with positive excess flow, and some vertices with zero excess flow. Also, we do not violate the capacity constraints. These properties define the notion of a *preflow*.

Definition 4 (Preflow) *An assignment of a non-negative flow $f(u, v)$ to each edge (u, v) of a network $(G = (V, E), s, t, c)$ is a preflow if*

- *for each edge (u, v), $f(u, v) \leq c(u, v)$*
- *for each vertex $v \in V - \{t\}$,*

$$\sum_u f(u, v) - \sum_w f(v, w) \geq 0$$

A preflow in which all excess flows $e_f(v)$ are zero for each $v \in V - \{s, t\}$ is a feasible flow.

The definition of *residual network* for a preflow is the same as for a flow; the capacity of an edge (u, v) in the residual network is

$$c(u, v) - f(u, v) + f(v, u)$$

If the edge (u, v) has capacity $\geq r$ in the residual network corresponding to a preflow f, and the vertex u has excess flow $\geq r$, then we can send r units of flow from u to v (by increasing $f(u, v)$ and/or reducing $f(v, u)$) and create another preflow. In the new preflow, the excess of u is r units less than before, and the excess flow of v is r units more than before.

Such a "shifting" of excess flow from one node to another is the basic operation of a push-relabel algorithm, and it is called a *push* operation. If we push an amount of flow along an edge equal to its capacity in the residual network, then we call it a *saturating push*, otherwise we call it a *nonsaturating push*. We execute a push operation provided that we find a pair of vertices such that we can push from a "higher" vertex to a "lower" vertex, according to the height function $h[\cdot]$.

If the above operation is not possible, we take a vertex with excess flow, and we increase its height. This operation is called a *relabel* operation.

The reader should try running this algorithm by hand on a few examples to get a sense of how it works.

By our discussion so far, it is by no means clear that this algorithm terminates at all. Indeed, in the next lecture we will show that

- Each vertex can reach, at most, height $2|V|$, and so the maximum number of relabel operations that can be executed is $2|V|^2$;
- The number of saturating push that can be executed along an edge (u, v) is at most $|V|$, and so the total number of saturating push operations that can be executed is at most $2|E| \cdot |V|$. (There are up to $2|E|$ edges that can appear in the residual networks at various stages.)
- The total number of nonsaturating push operations that can be executed is at most $4|V|^2 \cdot |E|$. (This will be the only difficult part of the analysis.)
- Each operation can be executed in constant time.

So overall we have a running time $O(|V|^2 \cdot |E|)$.