COMS 4995-2: Advanced Algorithms (Spring'20) Feb 6, 2020

Lecture 6: Polynomial-time algorithms for max-flow

Instructor: *Alex Andoni* Scribes: *Ana Maria Martinez Gomez, Arjun Srivatsa*

1 Introduction

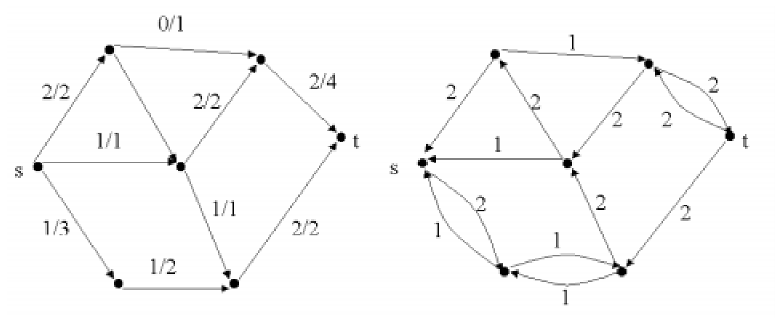
In the previous lecture we started talking about the max-flow problem and the Ford-Fulkerson Algorithm. Today's lecture is focused on the Proof of the Ford-Fulkerson Algorithm and Polynomial-time algorithms for max-flow. We start by reviewing briefly the concepts of the Residual Graphs/Augmenting Paths and Cuts.

2 Reminder: Residual Graphs

The residual graph, G_f of a graph G was defined in the last lecture as:

$$\begin{aligned} \forall\ (u,v) \in E: &\quad c_f(u,v) = c(u,v) - f(u,v) \\ \text{if } f(u,v) > 0: &\quad c_f(v,u) = f(u,v) \end{aligned} \tag{1}$$

Note that we need to consider some flow f to define the residual graph (It is only defined with an accompanying flow). Intuitively, this allows us to represent more "complex" paths which can either add to or remove from the flow of a corresponding edge. This is why in Ford-Fulkerson we work with residual graphs. For instance, if an edge from u to v has a flow 3 and capacity 5, we would then have in the residual graph an edge from u to v with flow 2 and an edge from v to u with flow 3. This intuitively means that we can add 2 more units in the u to v direction if we want to augment a path on this edge, or we can travel in the v to u direction by removing the 3 units of flow on the original graph. The figure below, from Williamson's lecture notes on network flow give a good example of a graph (left) and its corresponding residual graph (right).



3 Reminder: The Ford-Fulkerson Algorithm

Pulled from the prior scribe notes, the Ford-Fulkerson Algorithm is as follows:

1. Initialize: set $f(e) = 0$ for all $e \in E$, $G_f = G$.
2. While $\exists$ an $s \to t$ path P in G_f such that $\forall\ e \in P,\ c_f(e) > 0$:
   (a) $\delta(P) = \min_{e \in P} c_f(e)$
   (b) $\forall (u, v) \in P$:
   if $f(u, v) > 0$, then $f(v, u) = f(v, u) - \delta(P)$
   else, $f(u, v) = f(u, v) + \delta(P)$
   (c) Recompute G_f.

The idea behind this algorithm is to keep augmenting the flow from s to t in the residual graph. When we can no longer augment the path from s to t, we will prove that this flow is maximum.

4 Reminder: Cuts

A cut(or $s - t$ cut) is a separation of the set of vertices into two disjoint sets, S and $T = V - S$, where S contains the starting vertex s and T contains the sink vertex t. The capacity of a cut is defined as

$$c(S) = \sum_{u \in S, v \notin S} c(u, v)$$

The flow across a cut is defined as:

$$f(C) = \sum_{u \in S, v \notin S} f(u, v) - \sum_{u \in S, v \notin S} f(v, u)$$

. It is important to note that this concept of capacity only includes edges leaving the subset S and entering T. This is different from the flow across a cut, where we would include the edges going from T to S.

5 Proof of Ford-Fulkerson Algorithm

To complete the proof we will show that the following three statements are equivalent:

1. f is a max-flow
2. G_f the residual graph has no augmenting paths
3. G has a cut of value $c(S) = |f|$. (This is equivalent to the min-cut, max-flow theorem that says that the minimum cut will be equal to the maximum flow for a graph).

Remark: If we show that these three statements are equivalent, this means that the Ford-Fulkerson algorithm stops with the correct max flow because the algorithm only stops when the residual graph has no augmenting paths and then this would imply by 1.) that the flow has the correct max-flow. First, we

show that Statement 1 implies 2 then 2 implies 3 then 3 implies 1, which then proves the equality of all three statements.
1.) $\implies$ 2.):
We prove this by the contrapositive. First assume that 2.) is not true, that is our residual graph G_f has an augmenting path from s to t. That means that we can send more flow along some edge path from the source to the sink, which means we increase the maximum flow value by definition. So thus this implies that 1.) is false and 1.) $\implies$ 2.) by contrapositive.
2.) $\implies$ 3.):
We first state two lemmas:

Lemma 1. *For a given flow f, every cut of a graph G has the same flow value for the cut. That is $\forall C \in G$, where C is a valid cut, f(C) = $|f|$*

Lemma 2. *For any $s-t$ cut S, and any flow f, $|f| \leq c(S)$*

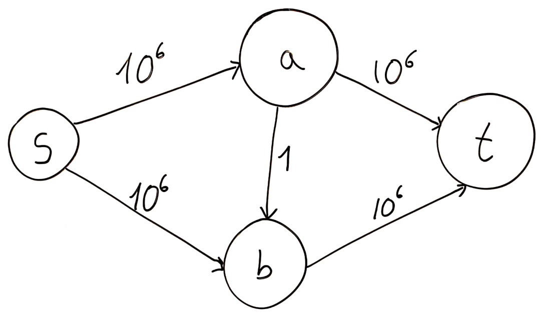
Assuming that the residual graph G_f has no augmenting paths, that means that there is some subset of vertices S from which we cannot traverse from S to T (where $T = V - S$). We show that this $s-t$ cut, S, in the original graph G has a capacity value of $|f|$. If we have that some edge $(u,v) \in G$ but $(u,v) \notin G_f$, this implies by the way we defined the residual graph that $f(u,v) = c(u,v)$ (Otherwise if f were not equal to c we would have a positive value on some edge and this would no longer be a desired disjoint cut). So hence for forward edges in the original graph corresponding to this $s-t$ cut in the residual graph, we necessarily have that the flow across the forward edges is equal to the capacity. Similarly, for edges from T to S, we must necessarily have that the flow across these edges is equal to zero. So, hence using this particular $s-t$ cut,S, we then have $c(S) = |f|$ since by lemma 1 the flow through S is equal to $|f|$, but in this case the flow is the capacity since all back edges are 0 and all the forward edges max the capacity. This is the statement of 3.) so we are done.
3.) $\implies$ 1.):
This immediately follows from lemma 2. If we have a flow with the value of a capacity of a cut, it must by definition be the maximum possible flow, since all other flows must be less than or equal to the value of this flow. The proof for lemma 2 was not rigorously provided in class but can be found on page 1-13 of Williamson's Cornell lecture notes.

6 Polynomial-time algorithms for max-flow

The presented Ford–Fulkerson algorithm runs in $O(|f|^*(m+n))$ (assuming the capacities are integers, as we send at least one unit of flow in every iteration).

Remark: With irrational flow values, it is not guaranteed that the algorithm terminates.

We can come up with graphs where the algorithm performs very badly, because of the election of the paths. For example:



In this graph, we take the following sequence of augmenting paths: s-a-b-t, s-b-a-t, s-a-b-t, ... (iterate between s-a-b-t and s-b-a-t). Then it takes $2 * 10^6$ steps to find the max flow. If we had instead taken the following sequence, we would have found it in only two steps: s-a-t, s-b-t. Selecting better augmenting paths helps us achieving our goal of getting a polynomial time algorithm.

6.1 Ideas for selecting augmenting paths

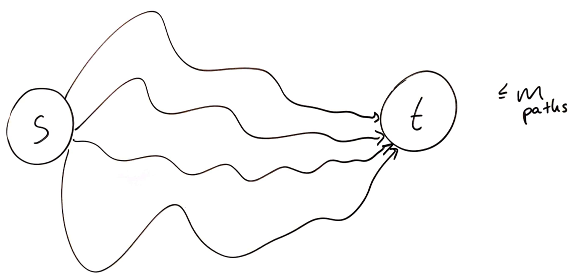
- Max capacity paths
- Randomized Paths
- Shortest paths
- Paths of near max capacity

6.2 Choose the path of the highest capacity

In every iteration, we select the path P for which $\delta(P)$ is maximized.

How do we find the path with the highest capacity? Trying all of them is not polynomial, but we can do a binary search in the edge capacities. If we know the value, we remove the edges which are smaller and look for a path.

What kind of progress do I make? Remember the flow decomposition we saw in the previous class: Any flow can be decomposed into m paths and $(n + m)$ cycles. Then we know that there exists a max flow which consists of at most m paths.



One of these paths has value at least $\frac{f^*}{m}$ (by averaging), where f^* is the value of max flow. That means that there exists a path P in G where $c(u, v) \geq \frac{f^*}{m}$. The path for which $\delta(P)$ is maximized has $\delta(P) \geq \frac{f^*}{m}$.

In every iteration, we get $\frac{1}{m}$ of the remainder flow closer to the optimum. This means that in each iteration we make less progress than in the previous one. It seems that the needed number of iterations is ∞. But the capacities are integers, so we only need to get to $f^* - 1$. So, after m iterations:

$$\leq f^*(1 - \frac{1}{m})^m \approx \frac{f^*}{e}$$

This means that after m iterations, we are halfway there. Then after $\Theta(m \log f^*)$ iterations, we are within 1. Using that $f^* \leq nC$, where C is the biggest capacity, we get that the number of iterations is $O(m \log nC)$. Then the running time of the algorithm is $O(m^2 \log C \log nC)$.

6.3 Next thing to do

Regarding the complexity, we want to:

- **Decrease the time**

- **Strongly polynomial:** Time only depends on m and n. C is normally not that big, but it could be causing the algorithm to be slower than expected.We can get $O(nm)$, or we can also get something like $O(\sqrt{n}m \log nC \log -)$.

The cleaner strongly polynomial result is using shortest augmenting paths. We will prove next time the following:
If $d(v,t) =$ distance in G_f from v to t, then for any v, $d(v,t)$ is non-decreasing over time.