601.433/633 Introduction to Algorithms   **Lecturer:** Michael Dinitz
Topic: Max-Flow II: Bipartite Matching, Edmonds-Karp   **Date:** 11/2/21

19.1   Introduction

Last time we analyzed the Ford-Fulkerson algorithm: choose an arbitrary path in the residual graph (known as an augmenting path), push as much flow along it as we can (whatever the "bottleneck capacity" of the path is), and repeat until there are no more $s \to t$ paths in the residual graph. Recall that after each step the residual graph changed. We showed that if all capacities are integers then a simple inductive proof shows that the max-flow we find will also be integral (the flow value on every edge will be an integer). We also showed that if all capacities are integers and F is the value of the maximum flow, then Ford-Fulkerson needs at most F iterations (and might in fact need that many – see the bad example from last time) so the total running time would be $O(Fm)$ (assuming that $m \geq n$). This is exponential in the input size, so FF is not a particularly efficient algorithm if there are large capacities.

Today we'll do two things: start with a very important application of max-flows for which FF is actually good enough, and then show how to choose augmenting paths carefully in order to guarantee that the running time of FF is actually pretty good in general.

19.2   Maximum Bipartite Matching

A *bipartite graph* $G = (V, E)$ is a graph with the property that V can be partitioned into two pieces L and R so that every edge goes between the two sides, i.e., $|e \cap L| = |e \cap R| = 1$ for all $e \in E$. A *matching* is a subset $M \subseteq E$ such that $e \cap e' = \emptyset$ for all $e, e' \in M$ with $e \neq e'$, i.e., no two endpoints share an edges. It is of course easy to find a matching (just return the empty set!), but an important algorithmic problem is to find the *maximum* matching in a bipartite graph. This has many applications, but you can think of it as an "assignment". For example, suppose that you're trying to schedule a meeting with your advisor during advising week. Then we can let L be the students and R be the possible timeslots of your advisor, and we put an edge between $u \in L$ and $v \in R$ if student u is able to make time v. Then a maximum matching is a schedule which maximizes the number of students who actually get to have a meeting.

Consider the following graph, for example:
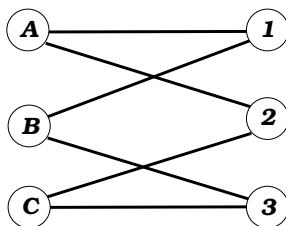
One maximum matching in this graph would be $\{\{A, 2\}, \{B, 1\}, \{C, 3\}\}$. Note that if we use a pure greedy-style algorithm, we might get stuck: if we first pick $\{A, 1\}$ and then add $\{C, 3\}$, there would be nowhere for us to match B.

It turns out that we can efficiently compute bipartite maximum matchings using max flow! To see how, let's turn this into a flow problem by changing the graph. First, we will make it directed by directing every edge from L to R (left to right). Then we will add a new source node s and an edge from s to every node in L, and a new sink node t and an edge from every node in R to t. Finally, we give every edge capacity 1. We now compute a maximum flow in this graph using Ford-Fulkerson, and return the matching M consisting of all $L - R$ edges with nonzero flow.

Theorem 19.2.1 M *is a maximum matching.*

Proof: Let's first prove that M is a matching. Since all capacities are 1, we know from the integrality property that the flow f returned by Ford-Fulkerson is integral, so every edge gets flow 0 or 1. So every edge in M has flow 1, and all other $L - R$ edges have flow 0. To see that M is a matching, note that there cannot be two edges in M incident on the same node $u \in L$, since the total capacity into u is only 1 and thus there can be at most one flow outgoing from u, and so only one edge in M incident on u. Similarly, there cannot be two edges in M incident on the same node $v \in R$, since the total outgoing capacity from v is only 1 and so there can be only one unit of incoming flow. Thus M is a matching.

To prove that it is maximum, note that $|M| = |f|$, i.e., the size of M is equal to the value of the flow f. If there were some larger matching M', then there would be a flow f' obtained by setting the flow on every edge in M' to 1, the flow of any (s, u) edge where u is matched in M' to 1, and the flow of any (v, t) edge where v is matching in M' to 1. This would be a valid flow with $|f'| = |M'| > |M| = |f|$, so would contradict f being a maximum flow. ∎

Theorem 19.2.2 *There is an algorithm to find a maximum bipartite matching in $O(mn)$ time.*

Proof: The algorithm above computes maximum matching. The running time is just the time to create the extra nodes and directions in the graph ($O(m)$), and then run Ford-Fulkerson. The maximum value flow F is equal to the maximum matching size, which is at most $n/2$, and hence the total running time is $O(mn)$. ∎

19.3   Edmonds-Karp

Now we want to design a polynomial-time algorithm to compute the maximum flow. We'll take the obvious approach: modify Ford-Fulkerson to choose more "reasonable" augmenting paths. We'll talk about two different definitions of "reasonable".

19.3.1   Edmonds-Karp 1

The most obvious thing to do is the be "greedy": let's choose the augmenting path which let's us push the most flow, i.e., increase the value of our flow by as much as possible. Recall that for an augmenting path P, we let $\min_{e \in P} c_f(e)$ be the minimum residual capacity on the path, and showed that we can increase our flow by this amount. So this suggests an obvious choice for a path:

choose the path P which maximizes the bottleneck capacity, i.e., choose

$$\underset{\text{augmenting paths } P}{\arg\max} \ \min_{e \in P} c_f(e).$$

This is sometimes called the "widest path." We define the capacity of a path to be its bottleneck capacity.

Choosing this path in the FF algorithm is the "Edmonds Karp #1" algorithm. What happens if we do this? What kind of running time do we get? Intuitively, we think that we're doing OK because we're always pushing as much flow as we can. It seems like this might be a problem if some bad choices in previous iterations lead to us not being able to push much flow anymore. So we want to prove that this can't happen: there is always some path which allows us to push a lot of flow.

Lemma 19.3.1 *In a graph with maximum $s-t$ flow F, there exists a path from s to t with capacity at least F/m*

Proof: Let $X = \{e : c(e) < F/m\}$ be the set of edges with capacity less than F/m. Suppose that in $G \setminus X$ there is no path from s to t. Then X is a cut with capacity at most $\sum_{e \in X} c(e) < m \cdot (F/m) = F$. Thus the max-flow in G is less than F, which contradicts the lemma statement. Thus in $G \setminus X$ there must still be some path P from s to t. Since P has no edges from X, every edge in P has capacity at least F/m. Thus P has capacity at least F/m. $\blacksquare$

This will now let us analyze the Edmonds-Karp #1 algorithm. Let F denote the actual maximum flow in G.

Theorem 19.3.2 *The Edmonds-Karp#1 algorithm terminates after at most $O(m \log F)$ iterations, if all capacities are integral.*

Proof: Lemma 19.3.1 tells us that we can always find a path of capacity at least F/m, so you might think that it would only take m iterations. However, there is a subtlety: F is the max-flow in the residual graph, not the starting graph! That is, we're actually finding paths in the residual graph, not the original graph, so the value of "F" is really the max-flow in the residual graph, i.e., the "amount of flow left to send".

So the right interpretation of Lemma 19.3.1 is that in each iteration, we decrease the "remaining flow to send" by a factor of $(1 - 1/m)$. After one iteration of the algorithm the remaining flow to send is at most $(1 - 1/m)F$, after two iterations it is at most $(1 - 1/m)^2 F$, etc. Since all capacities are integral, we know that everything stays integral throughout the algorithm, so if the remaining flow to send is less than 1 then it must equal 0 and we are done. So we will finish by iteration i where i is the smallest number such that $(1 - 1/m)^i F < 1$.

To solve this, we're going to use an enormously useful inequality which if you don't know, you should remember:

$$1 + x \leq e^x \qquad \text{for all } x \in \mathbb{R}.$$

This inequality might seem innocuous, but it's crazy how useful it is. Here, it allows us to prove that $i \leq m \ln F + 1$, since we get that

$$F(1 - 1/m)^{m \ln F} \leq F \left(e^{-1/m} \right)^{m \ln F} = F e^{-(m \ln F)/m} = F e^{-\ln F} = F/F = 1.$$

This implies the theorem. ■

There's one more subtlety: how can we actually find the widest path in the residual graph? It turns out that this can be done quickly using a variant of Dijkstra's algorithm in $O(m \log n)$ time (the same as Dijkstra). I won't go through the details, but it's a useful exercise to do at home. This means that the total running time of the algorithm is $O(m^2 \log n \log F)$. It's actually possible to get rid of the $\log n$ by being a little more clever, giving an algorithm with time $O(m^2 \log F)$.

19.3.2 Edmonds-Karp 2

Something might bother you about the previous algorithm: the $\log F$. While this is technically polynomial time, since F is at most exponential in the input size, it's not very "nice". Technically, this is called "weakly polynomial": it's polynomial, but it depends on the actual numbers. It would be nice if we could design a "strongly polynomial" algorithm: an algorithm with running time that doesn't depend on the numbers at all. This also means that it will be easier for us to handle non-integral capacities.

What choice of augmenting path will let us do this? It turns out that the right thing to do is to ignore the capacities completely, and just choose the *shortest* path in the residual graph (where each edge has length 1). It might seem weird to completely ignore capacities, since we might choose paths that only let us push a tiny amount of flow, but there's another sense in which it's pretty intuitive: if we're trying to design an algorithm whose running time doesn't depend on the capacities, then it's reasonable for our algorithm to not depend on capacities!

This algorithm, where we choose the *shortest* augmenting path, is the Edmonds-Karp #2 algorithm. Often people don't even talk about the first algorithm, and just call this the "Edmonds-Karp algorithm".

Theorem 19.3.3 *There are at most mn iterations in Edmonds-Karp #2.*

Proof: This argument is simple, but is quite subtle and somewhat tricky.

We'll prove this by proving the following claim: the distance from s to t goes up by at least 1 every m iterations. This immediately implies the theorem, since the distance is initially at least 1, and it can increase at most n times (if the distance is at least $n + 1$ then it is actually ∞, i.e., there is no path from s to t and the algorithm terminates).

To see this, consider some point in time and let d be the current distance from s to t in the residual graph. Lay out the residual graph in levels according to a BFS from s, i.e., nodes at level i are distance i away from s. Let's keep this layout fixed, and see what happens as we find augmenting paths and push flow along them. As long as the paths we find use only forward edges (edges from one layer to the next layer), each iteration causes at least one forward edge to be saturated and removed from the residual graph, and only backwards edges get added. Thus d does not decrease (since only backwards edges get added), and as long as d is unchanged we remove at least one forward edge per iteration. Hence d has to change (go up by at least 1) after at most m iterations.

This implies the claim and thus the theorem. ■

19.3.3 Extensions

Edmonds-Karp is not the only (or even the best) way of designing max-flow algorithms – there are more advanced methods such as Dinitz's algorithm (Yefim Dinitz, not me!) which essentially use many paths at once (these are called *blocking flow* methods), and even fancier methods known as *push-relabel* or *preflow-push* algorithms which are fundamentally different. We're not going to talk about these fancier methods, though – Edmonds-Karp is pretty good, and at the end of the day it's more important that you understand how to *use* max-flow algorithms than the details of the most state of the art algorithms.

There are also many important extensions, much as min-cost max-flow, which are similar to max-flow but are not the same. I probably won't talk about min-cost max-flow, but it's in the book and I'd definitely recommend looking at it.