1 Overview

The Ford-Fulkerson algorithm discussed in the last class takes time $O(mF)$, where F is the value of the maximum flow, when all capacities are integral. This is fine if all edge capacities are small, but if they are large numbers written in binary then this could even be exponentially large in the description size of the problem. In this lecture we examine some improvements to the Ford-Fulkerson algorithm that produce much better (polynomial) running times. We then consider a generalization of max-flow called the min-cost max flow problem. Specific topics covered include:

- Two faster max flow algorithms by Edmonds & Karp
- The min-cost max flow problem and algorithms for it

2 Network flow recap

Recall that in the maximum flow problem we are given a directed graph G, a source s, and a sink t. Each edge (u, v) has some capacity $c(u, v)$, and our goal is to find the maximum flow possible from s to t.

Last time we looked at the Ford-Fulkerson algorithm, which we used to prove the maxflow-mincut theorem, as well as the integral flow theorem. The Ford-Fulkerson algorithm is a greedy algorithm: we find a path from s to t of positive capacity and we push as much flow as we can on it (saturating at least one edge on the path). We then describe the capacities left over in a "residual graph" and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. Remember, one of the key but subtle points here is how we define the residual graph: if we push f units of flow on an edge (u, v), then the residual capacity of (u, v) goes down by f but also the residual capacity of (v, u) goes *up* by f (since pushing flow in the opposite direction is the same as reducing the flow in the forward direction). We then proved that this in fact finds the maximum flow.

Assuming capacities are integers, the basic Ford-Fulkerson algorithm could make up to F iterations, where F is the value of the maximum flow. Each iteration takes $O(m)$ time to find a path using DFS or BFS and to compute the residual graph. (We assume that every vertex in the graph is reachable from s, so $m \geq n - 1$.) So, the overall total time is $O(mF)$.

This is fine if F is small, like in the case of bipartite matching (where $F \leq n$). However, it's not good if capacities are in binary and F could be very large. In fact, it's not hard to construct an example where a series of bad choices of which path to augment on could make the algorithm take a very long time: see Figure 1.

Can anyone think of some ideas on how we could speed up the algorithm? Here are two we can prove something about.

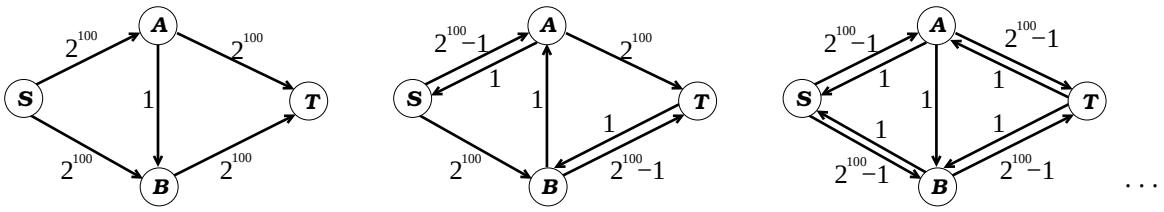

Figure 1: A bad case for Ford-Fulkerson. Starting with the graph on the left, we choose the path s-a-b-t, producing the residual graph shown in the middle. We then choose the path s-b-a-t, producing the residual graph on the right, and so on.

3 Fattest Augmenting Path (Edmonds-Karp #1 – Optional)

The first algorithm we study is due to Edmonds and Karp.[1] In fact, Edmonds-Karp #1 is probably the most natural idea that one could think of. Instead of picking an *arbitrary* path in the residual graph, let's pick the one of largest capacity. (Such a path is called a "maximum bottleneck path.") Can you think of an algorithm for this problem? Actually, you've probably already seen exactly such an algorithm before, but you may not have realized it. It turns out that paths in a *maximum spanning tree* are always maximum bottleneck paths, so Prim's algorithm, or your favorite MST algorithm can be used. How fast is such an algorithm? Let's see.

Claim 1 *In a graph with maximum s-t flow F, there must exist a path from s to t with capacity at least F/m.*

Can anyone think of a proof?

Proof: Suppose we delete all edges of capacity less than F/m. This can't disconnect t from s since if it did we would have produced a cut of value less than F. So, the graph left over must have a path from s to t, and since all edges on it have capacity at least F/m, the path itself has capacity at least F/m. ■

Claim 2 *Edmonds-Karp #1 makes at most $O(m \log F)$ iterations.*

Proof: By Claim 1, each iteration adds least a $1/m$ fraction of the "flow still to go" (the maximum flow in the current residual graph) to the flow found so far. Or, equivalently, after each iteration, the "flow still to go" gets reduced by a $(1 - 1/m)$ factor. So, the question about number of iterations just boils down to: given some number F, how many times can you remove a $1/m$ fraction of the amount remaining until you get down below 1 (which means you are at zero since everything is integral)? Mathematically, for what number x do we have $F(1 - 1/m)^x < 1$? Notice that $(1-1/m)^m$ is approximately (and always less than) $1/e$. So, $x = m \ln F$ is sufficient: $F(1-1/m)^x < F(1/e)^{\ln F} = 1$. ■

Now, remember we can find the maximum bottleneck path in time $O(m \log n)$, so the overall time used is $O(m^2 \log n \log F)$. You can actually get rid of the "$\log n$" by being a little tricky, bringing this down to $O(m^2 \log F)$.[2]

[1] Jack Edmonds is another of the greats in algorithms—he did a lot of the pioneering work on flows and matchings, and is one of the first people to propose that efficient algorithms should (at least) run in polynomial-time, instead of just stopping "in finite time". We've already met Dick Karp when discussing Karp-Rabin fingerprinting.

[2] This works as follows. First, let's find the largest power of 2 (let's call it $c = 2^i$) such that there exists a path from s to t in G of capacity at least c. We can do this in time $O(m \log F)$ by guessing and doubling (starting with

So, using this strategy, the dependence on F has gone from linear to logarithmic. In particular, this means that even if edge capacities are large integers written in binary, running time is polynomial in the number of bits in the description size of the input. Since the runtime still depends on the numbers in the input, such a runtime is called *weakly polynomial*.

We might ask, though, can we remove dependence on F completely and obtain a *strongly-polynomial time* algorithm? It turns out we *can*, using the second Edmonds-Karp algorithm.

4 Shortest Augmenting Paths (Edmonds-Karp #2)

The Edmonds-Karp #2 algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges), rather than the path of maximum capacity. This sounds a little funny but the claim is that by doing so, the algorithm makes at most mn iterations. So, the running time is $O(nm^2)$ since we can use BFS in each iteration. The proof is pretty neat too.

Claim 3 *Edmonds-Karp #2 makes at most mn iterations.*

Proof: Let d be the distance from s to t in the current residual graph. We'll prove the result by showing that (a) d never decreases, and (b) every m iterations, d has to increase by at least 1 (which can happen at most n times).

Let's lay out G in levels according to a BFS from s. That is, nodes at level i are distance i away from s, and t is at level d. Now, keeping this layout fixed, let us observe the sequence of paths found and residual graphs produced. Notice that so long as the paths found use only forward edges in this layout, each iteration will cause at least one forward edge to be saturated and removed from the residual graph, and it will add only backward edges. This means first of all that d does not decrease, and secondly that so long as d has not changed (so the paths *do* use only forward edges), at least one forward edge in this layout gets removed. We can remove forward edges at most m times, so within m iterations either t becomes disconnected (and $d = \infty$) or else we must have used a non-forward edge, implying that d has gone up by 1. We can then re-layout the current residual graph and apply the same argument again, showing that the distance between s and t never decreases, and there can be a gap of size at most m between successive increases.

Since the distance between s and t can increase at most n times, this implies that in total we have at most nm iterations. ■

This shows that the running time of this algorithm is $O(nm^2)$. An improvement to this algorithm called Dinic's algorithm makes it run in $O(n^2m)$ time, which is a big improvement for dense graphs.

$c = 1$, throw out all edges of capacity less than c, and use DFS to check if there is a path from s to t; if a path exists, then double the value of c and repeat). Now, instead of looking for *maximum* capacity paths in the Edmonds-Karp algorithm, we just look for s-t paths of residual capacity $\geq c$. The advantage of this is we can do this in linear time with DFS. If no such path exists, divide c by 2 and try again. The result is we are always finding a path that is within a factor of 2 of having the maximum capacity (so the bound in Claim 2 still holds), but now it only takes us $O(m)$ time per iteration rather than $O(m \log n)$.

5 Minimum-Cost Flows

We talked about the problem of assigning groups to time-slots where each group had a list of acceptable versus unacceptable slots. A natural generalization is to ask: what about preferences? E.g, maybe group A prefers slot 1 so it costs only \$1 to match to there, their second choice is slot 2 so it costs us \$2 to match the group here, and it can't make slot 3 so it costs \$infinity to match the group to there. And, so on with the other groups. Then we could ask for the *minimum-cost* perfect matching. This is a perfect matching that, out of all perfect matchings, has the least total cost.

The generalization of this problem to flows is called the *minimum-cost flow* problem. We are given a graph G where each edge has a *cost* $\$(e)$ in as well as a capacity $c(e)$[3]. We retain the capacity and flow conservation constraints from before, and continue to measure the *value* of a flow in the same way. The cost of a flow is then defined as the sum over all edges of the positive flow on that edge times the cost of the edge. That is,

$$cost(f) = \sum_{e \in E} \$(e) f(e).$$

Note importantly that the cost is charged *per unit* of flow on each edge. We are not paying a cost to "activate" an edge and then send as much flow through it as we like. This similar problem turns out to be NP-Hard. The goal of the minimum-cost flow problem is to find feasible flows of minimum possible cost. There are a few different variants of the problem.

- We will consider the **minimum-cost maximum flow** problem, where we seek to find the minimum-cost flow out of all possible maximum flows.
- An alternative formulation is to try to find the minimum-cost flow of value k for some given parameter k, rather than a maximum flow.

It is not hard to show that these problems are equivalent and easy to reduce to one another. There are also many other variants of the problem, but we won't consider those for now. Formally, the min-cost max flow problem is defined as follows. Our goal is to find, out of all possible maximum flows, the one with the least total cost. What should we assume about our costs?

- In this problem, we are going to allow *negative costs*. You can think of these as rewards or benefits on edges, so that instead of paying to send flow across an edge, you get paid for it.
- What about negative-cost cycles? It turns out that minimum-cost flows are still perfectly well defined in the presence of negative cycles, so we can allow them, too! Some algorithms for minimum-cost flows however don't work when there are negative cycles, but some do. We will be explicit about which ones do and do not.

Min-cost max flow is more general than plain max flow so it can model more things. For example, it can model the min-cost matching problem described above.

[3]There are a few different conventions for notation here. We're in an annoying spot since both cost and capacity start with c so we can't use it for both. In some sources, capacity is denoted by $u(e)$ and cost by $c(e)$, but we've already committed to $c(e)$ meaning capacity. I like this notation, which I've borrowed from Jeff Erikson's algorithms course at UIUC of using $\$(e)$ to denote the cost of an edge.

5.1 The residual graph for minimum-cost flows

Let's try to re-use the tools that we used to solve maximum flows for minimum-cost flows. Our key most important tool there was the *residual graph*, which we recall has edges with capacities

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E. \end{cases}$$

We will re-use the residual graph to attack minimum-cost flows. We need to generalize it, though, because our current definition of the residual graph has no ideas about the costs of the edges. For forward edges e in the residual graph (i.e., those corresponding to $e = (u,v) \in E$), the intuitive value to use for their cost is $\$(e)$, the cost of sending more flow along that edge. What about the reverse edges, though? That's less obvious. Let's think about it, when we send flow along a reverse in the residual graph, we are removing or *redirecting* the flow somewhere else. Since it cost us $\$(e)$ per unit to send flow down that edge initially, when we remove flow on an edge, we can think of getting a *refund* of $\$(e)$ per unit of flow – we get back the money that we originally paid to put flow on that edge. Therefore, the right choice of cost for a reverse edge in the residual graph is $-\$(e)$, the negative of the cost of the corresponding forward edge!

5.2 An augmenting path algorithm for minimum-cost flows

Now that we have defined a suitable residual graph for minimum-cost flows, we can build an algorithm based on augmenting paths in the same spirit as Ford-Fulkerson. If we just try to pick arbitrary augmenting paths, then it is unlikely that we will find the one of minimum cost, so how should we select our paths in such a way that the costs are accounted for? The most intuitive idea would be to select the *cheapest augmenting path*, i.e., the shortest augmenting path with respect to the costs as the edge lengths. This is not to be confused with the shortest augmenting path algorithm that selects the path with the fewest edges.

Since the edges are weighted by cost, a breadth-first search won't work anymore. How about our favorite shortest paths algorithm, Dijkstra's? Well, that won't quite work since the graph is going to have negative edge costs (note that even if the input graph does not have any negative costs, the residual graph will, so Dijkstra's will not work here). It turns out that there is a way to make Dijkstra's work, but it requires some extra fancy techniques that we won't have time to cover. So, let's use our next-favorite shortest paths algorithm that is capable of handling negative edges: Bellman-Ford! It is important to note here that since the algorithm makes use of shortest path computations, if there is a negative-cost cycle, this algorithm will not work.

Our algorithm is therefore the following. While there exists any augmenting path in the residual graph G_f, find the augmenting path with the cheapest cost and augment as much flow as possible along that path. Since this is just a special case of Ford-Fulkerson, the fact this algorithm finds a maximum flow is immediate, right? Well, not quite. Remember that since shortest paths only exist if there is no negative-cost cycle, we need to prove that our algorithm never creates one after performing an augmentation, or the next iteration's shortest path computation will fail.

Theorem 4 *At each iteration of the cheapest augmenting paths algorithm, the residual graph contains no negative-cost cycles.*

Proof: We can proceed by induction on the iterations of the algorithm. For the cheapest augmenting paths algorithm, we don't allow negative-cost cycles in G, therefore the initial residual graph,

which is just G, contains no negative-cost cycles, which establishes a base case. We now need to show that if at some step, the current residual graph contains no negative-cost cycles, then augmenting along a cheapest path produces a new residual graph that still contains no negative-cost cycles.

Consider the residual graph that contains no negative-cost cycles at the beginning of an iteration of the algorithm. We need to show that after augmenting along a cheapest augmenting path that there are still no negative-cost cycles in the residual graph. Suppose for the sake of contradiction that an augmentation introduces a negative cycle. Since the augmenting path creates reverse edges in the residual graph, at least one of these reverse edges must intersect the newly created negative-cost cycle. Suppose the corresponding edge e had cost $\$(e)$, and the remainder of the cycle has cost r. Since the cycle has a negative weight, we have $-\$(e) + r < 0$, but this implies that $r < \$(e)$, and hence it would have been a *cheaper path* to travel around the cycle instead of using the edge e! This is a contradiction, and hence we can conclude that the resulting residual graph has no negative cost cycles. ■

So, we have successfully proven that the algorithm will terminate with a maximum flow. What we still have to prove however is that this algorithm truly finds a *minimum-cost* flow.

5.3 An optimality criteria for minimum-cost flows

When studying maximum flows, our ingredients for proving that a flow was maximum were augmenting paths and the minimum cut. We'd like to find a similar tool that can be used to prove that a minimum-cost flow is optimal. First, let's be specific about what we mean by optimality. We will say that a flow f is *cost optimal* if it is the cheapest flow of all possible flows *of the same value*. That is to say we don't compare the costs of different flows if they have different values. Our goal is to find a tool to help us analyze the cost optimality of a flow. To do so, we're going to think about what kinds of operations we can do to modify a flow and change its cost *without* changing its value.

When finding maximum flows, our key tool was the augmenting path. If there existed an augmenting path in G_f, then by adding flow to it, we could transform a given flow into a more optimal one that was still feasible because adding flow to an s-t path preserved the flow conservation condition, and didn't violate the capacity constraint as long as we added an appropriate amount of flow. Therefore, the existence of an augmenting path proved that a flow was not maximum, and we were able to later prove that the lack of existence of an augmenting path was sufficient to conclude that a flow was maximum. We want to discover something similar for cost optimality of a flow. Augmenting paths were just one to to modify a flow while keeping it feasible. I claim that there is one other way that we can modify a flow while still preserving feasibility, but that also doesn't change its value. Instead of adding flow to an s-t path, what if we instead add flow to a *cycle* in the graph?

Since a cycle has an edge in and an edge out of every vertex, it should be clear that adding flow to a cycle in the residual graph preserves flow conservation, and hence feasibility. Furthermore, since the same amount of flow goes in and out of each vertex, the flow value is unaffected! However, adding flow to a cycle may in fact change the cost of the flow, which is what we wanted! Suppose the costs of the edges $e_1, e_2, ..., e_k$ on the cycle are $\$_1, \$_2, ..., \$_k$ for some cycle of length k. Then, adding Δ units of flow to the cycle will change the cost by

$$\sum_{i=1}^{k} \Delta \cdot \$_i = \Delta \cdot \sum_{i=1}^{k} \$_i.$$

Now, note that $\sum \$_i$ is just the weight of the cycle! So, we can say that if we add Δ units of flow to a cycle of weight W, then the cost of the flow changes by $\Delta \cdot W$. If $\Delta \cdot W$ is negative, then we have just shown that the cost of the flow *can be decreased*, so it wasn't optimal!

It turns out that this is the key ingredient for analyzing minimum-cost flows. Even better, we are getting a two-for-one deal because a lack of negative-cost cycles was what we already argued was required for our algorithm to produce a maximum flow. It turns out that this same condition allows us to prove that the flow is cost optimal.

Theorem 5 *A flow f is cost optimal if and only if there is no negative-cost cycle in G_f.*

Proof: We pretty much already proved the first direction in the previous paragraph. Suppose that there exists a negative cost cycle in the residual graph. Then by adding flow to this cycle, the value of the flow doesn't change, but the cost changes by $\Delta \cdot W$, where Δ is the amount of flow we can add, and W is the cost/weight of the cycle. Since W is negative, the cost decreases, and hence the flow f was not in fact cost optimal.

Now suppose that a flow f is not cost optimal. We need to prove that there exists a negative-cost cycle in its residual graph G_f. This case is much trickier. Since f is not cost optimal, there exists some other flow f' of the same value but which has a cheaper cost. Let's now consider the flow $f' - f$. Hang on, what does this even mean? Let's work that out. For each edge (u, v), consider two cases:

- if $f'(u, v) - f(u, v) \geq 0$, then we will say that the edge (u, v) has $f'(u, v) - f(u, v)$ flow,
- if $f'(u, v) - f(u, v) < 0$, then we will say that the *reverse* edge (v, u) has a flow of $f(u, v) - f'(u, v)$.

Okay, we've defined what $f' - f$ means, and I just told you that it is a flow, but how do we know that this is actually a valid flow, and what does it look like? Well, since both f' and f are feasible flows, they satisfy the flow conservation property, and hence their difference $f' - f$ also satisfies the flow conservation property (for example, if some vertex has 5 flow in and out in f' and 3 flow in and out in f, then their difference has 2 flow in and out). What about capacity constraints? Actually, it looks like this flow might not satisfy the capacity constraints of G, because it could send flow along a reverse edge that doesn't even exist in G. What actually matters to us is that $f' - f$ is a valid flow **in the residual graph** G_f.

If $f'(e) - f(e) \geq 0$, then the forward edge e has flow $f'(e) - f(e) \leq c(e) - f(e)$ which is the definition of the residual capacity $c_f(e)$. Alternatively, if $f'(e) - f(e) < 0$, then the reverse edge $\vec{e}$ has flow $f(u, v) - f'(u, v) \leq f(u, v)$, which is the definition of the residual capacity of $c_f(\vec{e})$. Therefore, $f' - f$ is a feasible flow in the residual graph! What is the value of this flow? By assumption, the values of f' and f are the same, hence their net flows out of s are the same. By taking the difference between the two, we can see that the net flow out of s in $f' - f$ is actually **zero**. So, we have a flow of value zero, but it isn't the all-zero flow, so what does this look like? Well, recall from our earlier discussion that a *cycle* of flow results in no net change to the value. We can also make this argument in the other direction and establish that a flow of value zero must be a *collection of cycles*. For this reason, such a flow is often called a *circulation*.

Lastly, we should think about the *cost* of this flow. Some simple algebra will show us that

$$\text{cost}(f' - f) = \text{cost}(f') - \text{cost}(f).$$

Since we assumed that $\text{cost}(f') < \text{cost}(f)$, this must be a negative cost. So, we have a collection of cycles of flow whose total cost is negative, therefore at least one of those cycles must have a negative cost. Since $f' - f$ is feasible in the residual graph, this negative-cost cycle exists in the residual graph, which concludes the proof. ■

Since we already argued in Theorem 4 that the cheapest augmenting path algorithm never creates a negative-cost cycle in the residual graph, this theorem proves that the resulting flow is also cost optimal. Therefore we can conclude that the algorithm successfully finds the minimum-cost maximum flow!

5.4 Runtime

We can argue about the runtime of the cheapest augmenting path algorithm using the same logic that we used for Ford-Fulkerson. At every iteration of the algorithm, at least one unit of flow is augmented, and every iteration has to run the Bellman-Ford algorithm which takes $O(nm)$ time. Therefore, the worst-case running time of cheapest augmenting paths is $O(nmF)$, where F is the value of the maximum flow. There are numerous ways to speed this up, some of which achieve polynomial time, but we won't have time to cover those for now.