The Ford-Fulkerson algorithm discussed in the last class takes time $O(F(n + m))$, 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:

- Edmonds-Karp Algorithm #1

- Edmonds-Karp Algorithm #2

- Further improvements

- Min-cost max flow

1   Network flow recap

Recall that in the network 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. (To reduce notation, let's assume we have pre-processed the graph to delete any disconnected parts so that $m \geq n - 1$.) 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.

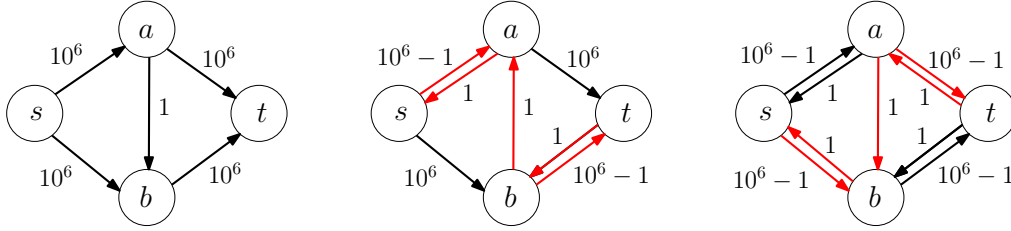Can anyone think of some ideas on how we could speed up the algorithm? Here are two we can prove something about.

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. Each time we push one unit of flow, and hence it will take two million iterations to achieve the max-flow. The red edges are those that are new, or have had their capacities changed.

2   Edmonds-Karp #1

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 (we call this the "maximum bottleneck path").

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 $F(1 - 1/m)^x < 1$? Recall that "one of the most useful inequalities ever" from the Min-Cut Lecture #4:

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

In fact, the inequality is strict except for the case when $x = 0$. So, $x = m \ln F$ is sufficient: $F(1 - 1/m)^x < Fe^{-x/m} = F(1/e)^{\ln F} = 1$. ∎

One detail remains: how do we find the maximum bottleneck path fast? It turns out that you can change Dijkstra's algorithm slightly, still keeping the run-time at $O(m \log n)$. (You should try to prove this for yourself! We won't cover this in lecture — if needed, we can post notes about it.) 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 $O(m^2 \log F)$.[1]

---

[1]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.

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

3 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. ∎

4 Further discussion: Dinic and MPM

Can we do this a little faster? (We may skip this depending on time, and in any case the details here are not so important for this course, but the high level idea is nice.)

The previous algorithm used $O(mn)$ iterations at $O(m)$ time each for $O(m^2 n)$ time total. We'll now see how we can reduce to $O(mn^2)$ time and finally to time $O(n^3)$. Here is the idea: given the current BFS layout used in the Edmonds-Karp argument (also called the "*level graph*"), we'll try in $O(n^2)$ time all at once to find the maximum flow that only uses forward edges. This is sometimes called a "*blocking flow*". Just as in the analysis above, such a flow guarantees that when we take

---

$c = 1$, 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 $O(m \log n)$.

the residual graph, there is no longer an augmenting path using only forward edges and so the distance to t will have gone up by 1. So, there will be at most n iterations for a total time of $O(n^3)$.

To describe the algorithm, define the *capacity of a vertex v* as $c(v) = \min[c_{in}(v), c_{out}(v)]$, where $c_{in}(v)$ is the sum of capacities of the in-edges to v and $c_{out}(v)$ is the sum of capacities of the out-edges from v. The algorithm is now as follows:

1. In $O(m)$ time, create the level graph G_{level} which is a BFS layout from s containing only the forward edges, where we then remove any nodes that can't reach t using these edges (e.g., by doing a backwards BFS from t and removing unmarked nodes). Compute the capacities of all nodes, considering just edges in G_{level}.

2. Find the vertex v of minimum capacity c. If it's zero, that's great: we can remove v and incident edges, updating capacities of neighboring nodes.

3. If c is not zero, then we greedily pull c units of flow from s to v, and then push that flow along to t. We then update the capacities, delete v and repeat. Unfortunately, this seems like just another version of the original problem! But, there are two points here:

   (a) Because v had *minimum* capacity, we can do the pulling and pushing in any greedy way and we won't get stuck. E.g., we can do a BFS backward from v, pulling the flow level-by-level, so we never examine any edge twice (do an example here), and then a separate BFS forward from v doing the same thing to push the c units forward to t.

   This right away gives us an $O(mn)$ algorithm for saturating the level graph ($O(m)$ per node, n nodes), for an overall running time of $O(mn^2)$. So, we're half-way there.

   (b) To improve the running time further, the way we will do the BFS is to examine the in-edges (or out-edges) one at a time, fully saturating the edge before going on to the next one. This means we can allocate our time into two parts: (a) time spent pushing/pulling through edges that get saturated, and (b) time spent on edges that we didn't quite saturate (at most one of these per node). In the BFS we only take time $O(n)$ on type-b operations since we do at most one of these per vertex. We may spend more time on type-a operations, but those result in deleting the edge, so the edge won't be used again in the current level graph. So over *all* vertices v used in this process, the *total* time of these type-a operations is $O(m)$. That means we can saturate the level graph in time $O(n^2 + m) = O(n^2)$.

Our running time to saturate the level graph is $O(n^2 + m) = O(n^2)$. Once we've saturated the level graph, we recompute it (in $O(m)$ time), and re-solve. The total time is $O(n^3)$.

This algorithm is due to Malhotra, Pramodh Kumar, and Maheshwari (and hence is called the MPM algorithm). Dinic's algorithm is also based on blocking flows, i.e., on saturating the level graph, but it slightly different.

By the way, even this is not the best running time known: for a long time, the best algorithms have performance of almost $O(mn)$ (and the best algorithms with no dependence on F still have run-time about $O(mn)$). The best algorithm with a dependence on F is that of Goldberg and Rao, and has a running time of $O(m \min(m^{1/2}, n^{2/3}) \log(n^2/m) \log F)$, and also based on finding blocking flows.

5 Min-cost Matchings, Min-cost Max 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 *min-cost max flow* problem. Formally, the min-cost max flow problem is defined as follows. We are given a graph G where each edge has a *cost* $w(e)$ in as well as a capacity $c(e)$. The cost of a flow is the sum over all edges of the positive flow on that edge times the cost of the edge. That is,

$$cost(f) = \sum_e w(e)f^+(e),$$

where $f^+(e) = \max[f(e), 0]$. Our goal is to find, out of all possible maximum flows, the one with the least total cost. We can have negative costs (or benefits) on edges too, but let's assume just for simplicity that the graph has no negative-cost cycles. (Otherwise, the min-cost max flow will have little disconnected cycles in it; one can solve the problem without this assumption, but it's conceptually easier with it.)

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.

There are several ways to solve the min-cost max-flow problem. One way is we can run Ford-Fulkerson, where each time we choose the *least cost* path from s to t. In other words, we find the shortest path but using the costs as distances. To do this correctly, when we add a back-edge to some edge e into the residual graph, we give it a cost of $-w(e)$, representing that we get our money back if we undo the flow on it. So, this procedure will create residual graphs with negative-weight edges, but we can still find shortest paths in them using the Bellman-Ford algorithm.

We can argue correctness for this procedure as follows. Remember that we assumed the initial graph G had no negative-cost cycles. Now, even though each augmenting path might add new negative-cost *edges*, it will never create a negative-cost *cycle* in the residual graph. That is because if it did, say containing some back-edge (v, u), that means the path using (u, v) wasn't really shortest since a better way to get from u to v would have been to travel around the cycle instead. So, this implies that the final flow f found has the property that G_f has no negative-cost cycles either. This means that f is optimal: if g was a maximum flow of lower cost, then $g - f$ is a legal circulation in G_f (a flow satisfying flow-in = flow-out at *all* nodes including s and t) with negative cost, which means it would have to contain a negative-cost cycle, a contradiction.

The running time for this algorithm is similar to Ford-Fulkerson, except using Bellman-Ford instead of Dijkstra. It is possible to speed it up, but we won't discuss that in this course.