1 Overview

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
- Dinic's Algorithm
- Min-cost max flow

2 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.

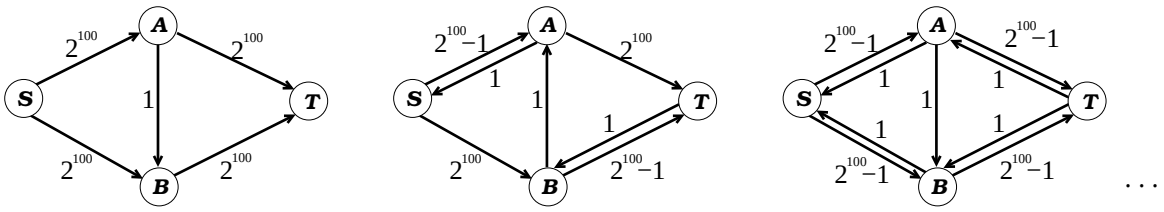


Figure 1: A bad case for Ford-Fulkerson. Starting with the graph on the left, we choose the path s-a-b-t, producing the residual graph shown in the middle. We then choose the path s-b-a-t, producing the residual graph on the right, and so on.

3 Edmonds-Karp #1

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." Dijkstra's algorithm can be trivially modified to find such paths. Instead of keeping tentative distance values in the priority queue, we keep maximum bottleneck values. The running time is the same as Dijkstra's.)

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 $c = 1$, 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

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.

4 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 $O(nm^2)$.

5 Dinic's Algorithm

We can improve this algorithm to $O(n^2m)$ by reorganizing the computation. This is called Dinic's algorithm[3].

---

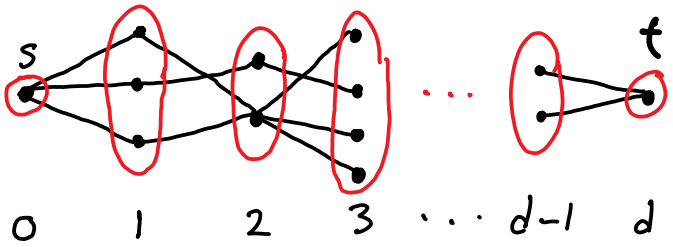
algorithm, we just look for s-t paths of residual capacity $\geq c$. The advantage of this is we can do this in linear time with DFS. If no such path exists, divide c by 2 and try again. The result is we are always finding a path that is within a factor of 2 of having the maximum capacity (so the bound in Claim 2 still holds), but now it only takes us $O(m)$ time per iteration rather than $O(m \log n)$.

[3]The idea of finding the blocking flow, i.e., maximum flow possible in the level graph G_{level}, and then iterating this as the distance from s to t gets bigger, is due to Yefim Dinitz. He gave this algorithm while getting his M.Sc. in Soviet Russia in 1969 under G. Adel'son Vel'sky (of AVL trees fame), in response to a exercise in an Algorithms class, and published it in 1970. The iron curtain meant this was not known in the west for a few more years, and then only as Dinic's [*sic*] algorithm. Dinitz talks about his algorithm and some history here. The algorithm we present here is due to V.M. Malhotra, M. Pramod Kumar, and S.N. Maheshwari (1978), and hence is called the *MPM Algorithm*. A similar idea was developed by Alexander Karzanov in Russia in 1974.

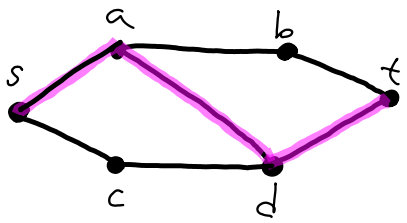
If we examine what E-K#2 does, we see that it generates a sequence of augmenting paths of non-decreasing length. What Dinic's algorithm does is it finds all the necessary augmenting paths of a given length with one *blocking flow* computation. It then augments the graph by this blocking flow, and repeats. The number of passes needed is at most $n-1$, because each one increases the length of the paths being searched for.

So at the beginning of each pass the algorithm spends $O(m)$ time to explicitly build the layered graph. This is done via a simple breadth-first-search starting from s.

The layered graph L only includes the edges that go from one level to the next higher level. The schematic below illustrates a layered graph. All edges go from left to right.



So the crux of the algorithm is to find a flow in L from s to t which is *blocking*, which means that every path from from s to t that steps from level to level (always increasing) has a saturated edge on it. Note that this is not the same as a maximum flow, as illustrated in the figure below.



In the above diagram all the edges go from left to right. Level 0 is comprised of $\{s\}$, level 1 is comprised of $\{a, c\}$, level 2 is $\{b, d\}$, and level 3 is $\{t\}$. All edges are of capacity 1, and directed from a level i to level $i+1$.

A flow of one unit along the highlighted path from left to right is a blocking flow. In the residual graph that results, there is no path of positive capacity from s to t that starts at level 0 and proceeds to level 1, 2, and then 3. However there *is a path* of positive capacity from s to t, namely $[s, c, d, a, b, t]$. This path will be discovered later when the algorithm is working on paths of 5 edges.

On the next page is the pseudo code for computing a blocking flow.

```
Dinic's Algorithm to find a blocking flow in a layered graph L.  The
edges of this graph only go from a layer to the next higher layer.

The graph is represented as follows:

degree[v] : the number of edges out from v to the next level.

N[v][i]   : The ith neighbor of vertex v.  (Of course these vertices are
            all in the next layer.)  The index i ranges from 0 to degree[v]-1.

Cap[v][i] : The capacity of the edge from v to N[v][i].
            These change over time as we push flow through.

The adjusted capacities are how the algorithm returns its
result.

The algorithm also makes use of the following variable:

C[v]      : The index of the "current" neighbor of v.
            N[v][C[v]] is that neighbor.  Initially this
            is 0, and eventually it could reach degree[v].

    Dinic(L) {
        Build the representation of L described above.
        flow <- 0;
        while(true) do{
            f <- Search(s);
            if f = 0 then return flow;
            else {
                flow = flow + f;
                Augment the current path from s to t by f.
                (i.e. reduce all these capacities by f)
                This path is obtained by simply following
                current edges.
            }
        }
    }

    Search(v) {
    // Search for a path from v to t of positive capacity.
    // If there is no such path, return 0.
    // As a side-effect, this may increase C[v]

        if v=t then return infinity;
        while(true) do {
           if C[v] = degree[v] then return 0; (call v "dead")
           if Cap[v][C[v]] = 0 then C[v] <- C[v] + 1
           else {
               f <- Search(N[v][C[v]])
               if f > 0 then return min (f, Cap[v][C[v]])
               else C[v] <- C[v] + 1
            }
        }
    }
```

Discussion of Correctness

Claim: once a vertex v is labeled "dead" then there is no path of positive capacity from v to t.

The proof is by induction. The only way a vertex is said to be dead is if every neighbor w on the next level either (1) gives a return value of zero on `Search(w)`, or (2) the edge from v to w has zero capacity. This shows that there is no path of positive capacity from v to t.

Discussion of Running Time

Each path augmentation causes the residual capacity of an edge to go to zero. This can happen at most m times. The work of all of these augmentations is at $O(nm)$.

What about the time of `Search()`? The running time is proportional to the total number of calls to the `Search()` function. Every time `Search()` returns a zero value, it causes `C[v]` (for some v) to increase. This can happen at most m times, in total. When it returns a positive value, this causes a cascading sequence of returns of positive value all the way back to the initial call of `Search(s)`. These calls to search can be paid for by the lengths of all the augmenting paths, which total to at most nm.

6 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.

6.1 A Ford-Fulkerson-like Algorithm

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 show it does not create negative cycles, so we can still find shortest paths in them using the Bellman-Ford algorithm.

Why don't we get negative cycles in the residual graph? Remember that we assumed the initial graph G had no negative-cost cycles. For sake of a contradiction, suppose an augmenting path added in negative-cost back-edges into the residual graph, and we created a negative cycle. This cycle must contain one of these back-edges (v, u). This 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. (You should be careful in this argument when the newly-created negative cycle may contain multiple back edges.) This gives the contradiction, which proves we do not create any negative-cost cycles by augmenting along least-cost s-t paths.

Hence the final flow f found has the property that the residual graph G_f has no negative-cost cycles either. We claim this means that f is optimal. Suppose 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, and respecting arc capacities), since the arc capacities in G_f are $c(u, v) - f(u, v)$. Moreover, $g - f$ has negative cost since g costs less than f. Since this circulation can be broken into a collection of cycles, it must contain at least one 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, to get an algorithm whose running time is like Edmonds-Karp #1, or even one whose runtime just depends on m and n, but we won't discuss that in this course.