Network Flows II

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 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 regardless of the value of the max flow or capacities.

We will then consider a generalization of max flow called *minimum-cost* flows, where edges have costs as well as capacities, and the goal is to find flows with low cost. This problem has a lot of nice properties, and is also highly practical since it generalizes the max flow problem.

Objectives of this lecture

In this lecture, we will:

- See an algorithm for max flow with polynomial running time (Edmonds-Karp)
- Define and motivate the *minimum-cost flow* problem
- Derive and analyze some algorithms for minimum-cost flows

Recommended study resources

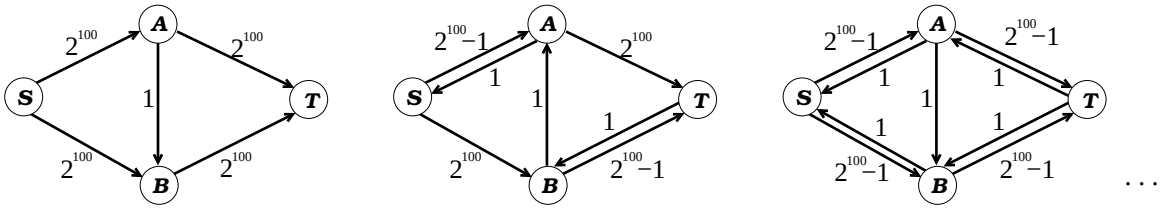
- CLRS, *Introduction to Algorithms*, Chapter 26, Maximum Flow
- DPV, *Algorithms*, Chapter 7.2, Flows in Networks
- Erikson, *Algorithms*, Chapter 10, Maximum Flows & Minimum Cuts

1 Network flow recap

Recall that in the maximum flow proble, 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 min-cut max-flow theorem, as well as the integrality theorem for flows. 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", which accounts for remaining capacity as well as

the ability to redirect existing flow (and hence "undo" bad previous decisions) and repeat the process, continuing until there are no more paths of positive residual capacity left between s and t. 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 large and F could be very large. Here's an example that could make the algorithm take a very long time. If the algorithm selects the augmenting paths $s \to A \to B \to t$, then $s \to B \to A \to t$, repeating..., then each iteration only adds one unit of flow, but the max flow is 2^{101}, so the algorithm will take 2^{101} iterations. If the algorithm selected the augmenting paths $s \to A \to t$ then $s \to B \to t$, it would be complete in just two iterations! So the question on our minds today is can we find an algorithm that provably requires only polynomially many iterations?



2 Shortest Augmenting Paths Algorithm (Edmonds-Karp)

There are several strategies for selecting better augmenting paths than arbitrary ones. Here's one that is quite simple and has a provable polynomial runtime. Its called the Shortest Augmenting Paths algorithm, or the Edmonds-Karp algorithm. The name of the algorithm might give away a slight hint of what it does.

Algorithm: Shortest Augmenting Paths (Edmonds-Karp)

Edmonds-Karp implements Ford-Fulkerson by selecting the *shortest* augmenting path each iteration.

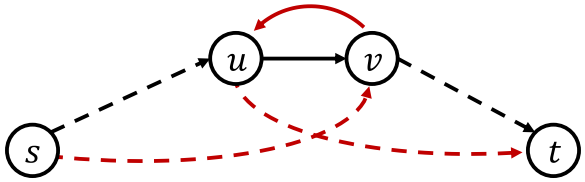
Unsurprisingly, the Shortest Augmenting Paths (Edmonds-Karp) algorithm works by always picking the *shortest* path in the residual graph (the one with the fewest number of edges). By example, we can see that this would in fact find the max flow in the graph above in just two iterations, but what can we say in general? In fact, the claim is that by picking the shortest paths, 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.

Theorem: Runtime of Edmonds-Karp

The Shortest Augmenting Paths algorithm (Edmonds-Karp) 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:

Claim (a): d never decreases Consider one iteration of the algorithm. Before adding flow to the augmenting path, every vertex v in G has some distance d_v from the source vertex s *in the residual graph.* Suppose the augmenting path consists of the vertices $v_1, v_2, \ldots v_k$. What can we say about the distances of the vertices? Since the path is by definition a *shortest path*, it must be true that $d_{v_i} = d_{v_{i-1}} + 1$, that is, every vertex is one further from s. Now perform the augmentation and consider what changes in the residual graph. Some of the edges (at least one) become *saturated*, which means that the flow on the edge reaches its capacity. When this happens, that edge will be removed from the residual graph. But another edge might appear in the residual graph! Specifically, when $e = (u, v)$ goes from zero to nonzero flow, $e' = (v, u)$ may appear in the residual graph as a back edge (if it doesn't exist already). Can this lower the distance of any vertex? No, $d_v = d_u + 1$, so adding an edge from v to u can't make a shorter path from s to t.



Therefore, since the distance to any vertex can not decrease, d can not decrease.

Claim (b): every m iterations, d has to increase by at least 1 Each iteration saturates (fills to capacity) at least one edge. Once an edge is saturated it can not be used because it will not appear in the residual graph. For the edge to become usable again, it must be the case that its back edge in the residual graph is used, which means that the back edge needs to appear on the shortest path. However, if $d_v = d_u + 1$, then it is not possible for the back edge (v, u) to be on a shortest path, so this can *only* occur if d increases. Since there are m edges, d must increase by at least one every m iterations.

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

This shows that the running time of this algorithm is $O(nm^2)$. Note that this is true for **any** capacities, including large ones and non-integer ones. So we really have a polynomial-time algorithm for maximum flow!

3 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. This was the *bipartite matching* problem. A natural generalization is to ask: what about preferences? E.g, maybe group A prefers slot 1 so it costs