Intro to LO, Lecture 8

Martin Böhm
University of Wrocław, Winter 2023/2024

Flows in networks

Recall the problem Maximum Flow from Lecture 1:

Input: An edge-weighted, directed graph $G = (V, E, c)$ (a network). Every directed edge has an associated capacity $c_e \geq 0$. Additionally, there are two special vertices – source s and sink t.

Output: Any flow – function $f : E \to \mathbb{R}_0^+$ which *obeys capacities* ($f_e \leq c_e$) and *follows Kirchhoff's law* – in every non-special vertex, fluid coming in = fluid coming out.

Goal: Find a flow that sends as much fluid as possible from s to t.

$$\max \sum_{\vec{si}} x_{\vec{si}}$$

$$\begin{aligned}
\text{s.t.}\quad & \forall \vec{ij} \in E(G): & x_{\vec{ij}} \leq c_{\vec{ij}} \\
& \forall v \in V(G) \setminus \{s,t\}: & \sum_{\vec{vi}} x_{\vec{vi}} - \sum_{\vec{jv}} x_{\vec{jv}} = 0 \\
& \forall \vec{ij} \in E(G): & x_{\vec{ij}} \geq 0, x_{\vec{ij}} \in \mathbb{R}
\end{aligned}$$

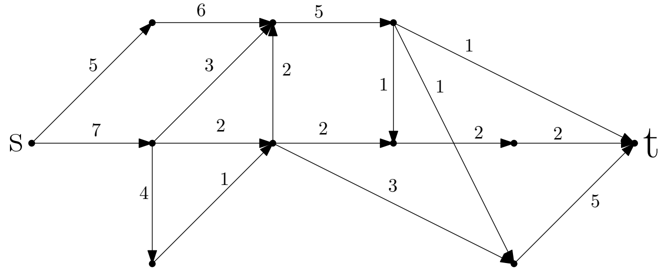


Figure 1: A sample instance for Maximum Flow.

T(Max-flow min-cut): In any network, the value of a maximum flow is equal to the capacity of a minimum cut.

P: Dualize the LP for maximum flow. The objective function of max flow is $0-1$ plus total unimodularity $\to$ the dual LP is also integral. Rework the LP of the dual and prove that it computes the minimum cut.

D(Saturated edge): For a given flow and edge, we say an edge e is *saturated* if $f_e = c_e$.

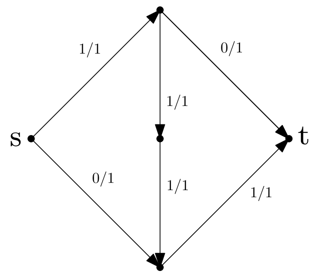
The Fake Ford-Fulkerson Algorithm

The idea: use a greedy approach to compute the maximum flow.

ALG (F F-F): Repeat as long as possible:

1. Find any $s-t$–augmenting path.
2. Send as much as possible.

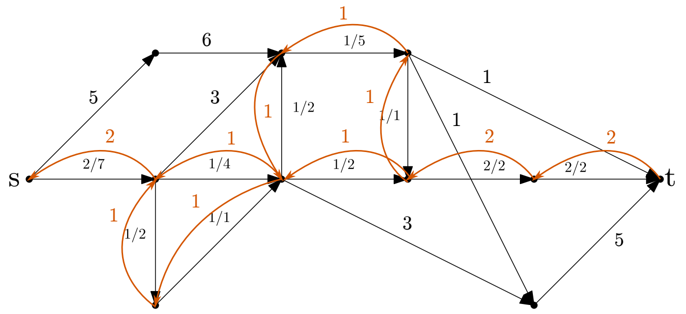
O: The Fake Ford Fulkerson algorithm will terminate and fail to find the maximum flow on some networks.



The proof to the example above. 1/1 means the flow value is 1 out of total capacity 1.

The Ford-Fulkerson Algorithm

D(Residual/Reserve network): When there is already flow in your network, a *residual network* (sometimes also called *reserve network*) creates, for any directed edge uv with flow value f_{uv}, an opposite edge vu with capacity f_{uv}. Using this edge corresponds to "rerouting" the flow.



An visualization of a single flow and its residual edges in the network.

ALG (F-F): Create the residual network. Repeat as long as possible:

1. Find any $s-t$–augmenting path (for example, via DFS).
2. Send as much as possible. Update the residual network.

T: If the Ford-Fulkerson algorithm terminates, it returns a maximum flow.

P: Take all reachable vertices from s by augmenting paths, let us call it the S-part. All edges going out of the S-part are saturated. There is inflow 0 on all edges incoming to the S-part – otherwise residual edges allow us to extend the S part.

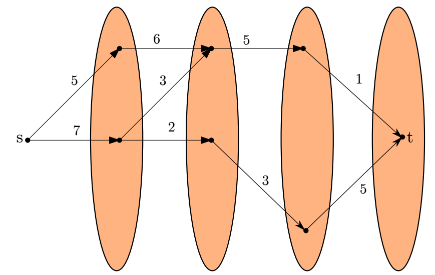
Since there is inflow 0 to the S-part, the value of the cut = value of the outgoing edges = value of the flow from s to t. We found a cut of the same size as the flow.

The Dinitz Algorithm

D(Blocking flow of distance d): We say that a flow is a *blocking flow* for distance d if it is true that for all s,t-paths in the residual network of distance at most d, there exists one saturated edge.

Main idea: Add augmenting flows, not just augmenting paths. The Dinitz algorithm iteratively updates the current flow by blocking flows of distance d.

1. Start with any flow f, e.g. empty (zero everywhere).
2. Iteratively improve the flow by augmenting flows: (outer loop)
   - (a) Construct a residual network: vertices and edges are the same, capacities are determined by the reserves in the original network. In the following we work with that.
   - (b) Find the shortest st-path. If none exists, we stop.
   - (c) We clean the network to make it into a *layer network*, i.e., we leave only vertices and edges on the shortest *st*-paths.
   - (d) Find the blocking flow f_B in the layer network:
   - (e) $f_B \leftarrow$ empty flow
   - (f) Adding st-paths one by one: (inner loop)
     - i. We find the st-path. E.g. greedily.
     - ii. Send as much as possible along the discovered path.
     - iii. Delete saturated edges. (Beware, deleting edges can create dead ends, which will pollute the network and make greedy pathfinding impossible.)
     - iv. Clean the network.
   - (g) Improve f by f_B



The first iteration of the layer network for the Dinitz algorithm for the network on Figure 1.

O: Cleanup takes only $O(m)$ time per iteration.

O: Searching for blocking flow takes $O(nm)$.

P: We can find one path in the layer network in time $O(n)$, because every step moves us one layer forward and there is no backtracking. After one path is taken, at least one edge is saturated and leaves the layer network for now. Thus, $O(nm)$ running time.

O: The number of iterations for searching for the blocking flow is $O(n)$.

P: The only new edges that get added in the next residual network are backwards edges, which only increase the shortest $s-t$ length. Plus (after cleanup) longer paths may become relevant, but still, shortest $s-t$ path increases before every reserve network construction, and so we have $O(n)$.

Malhotra-Kumar-Maheswari

A modification of Dinitz that runs in $O(n^3)$ time. The key is to do the search for blocking flow (outer loop) faster.

ALGM-K-M: For every vertex, compute the *out-reserve* $r^-(v)$ as the sum of the (reserve) capacities of the outedges, *in-reserve* $r^+(v)$ analogously, and $r(v) = \min(r^+(v), r^-(v)$.

Maintain the numbers throughout the search. Take the minimum of $r(v)$ in the network, and send the full value first towards t, saturating edge by edge (only at most one unsaturated) and then do the same backwards towards s.

Every edge either fully leaves the system ($O(m)$) or is the unsaturated one (at most two per vertex) $\rightarrow O(n^2)$ running time for finding the blocking flow.

Exercises

EXERCISE ONE (A similar thing from the last exercise session, done differently.) Using an algorithm for maximum flow, prove that one can find maximum matching in a bipartite graph in polynomial time.

EXERCISE TWO Suppose that we have an *undirected* graph G with s and t. Can you precisely specify how to use our knowledge of maximum flows to compute the s-t maximum flow in such a graph?

EXERCISE THREE Suppose that all capacities in the network are 1. (This happens for example when we wish to compute maximum matching via flows.) Can you improve the analysis of the Dinitz algorithm to show that it runs in $O(nm)$?

EXERCISE FOUR (An important exercise!) Kőnig's theorem states the following: In every bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover.

Prove Kőnig's theorem. Actually, there are at least two ways you can do it quickly – either directly via duality and unimodularity, or using flow networks.

EXERCISE FIVE The following set of ideas pushes the running time for unit capacities from $O(nm)$ to $O(m^{3/2})$. This only helps for sparse graphs, but following this idea further (which we will not do today) actually leads to an improved analysis of $O(n^{2/3}m)$.

1. Suppose that we are in the k-th iteration of the "outer loop" of the Dinitz algorithm. Thus, the shortest s-t-path in the residual network is of length at least k. Can you prove that there exists a directed s-t-cut in the residual network of size $\leq m/k$?
2. If we find a cut of size m/k, how many iterations of the "outer loop" are remaining?
3. How can we conclude that the running time is $O(m^{3/2})$?

EXERCISE SIX Suppose that all capacities are integral and bounded by some constant C (and the graph is not a multigraph). Can you improve the analysis of the Dinitz algorithm (without changing it) to show that the overall runtime is $O(Cn^2 + mn)$?