In this lecture, we take a look at the maximum flow problem, which we already mentioned in the first lecture. We consider the very simple Ford-Fulkerson Algorithm and then give the improved Edmonds-Karp Algorithm. In further lectures, we give more advanced algorithms and generalize this problem.

1 Maximum Flows

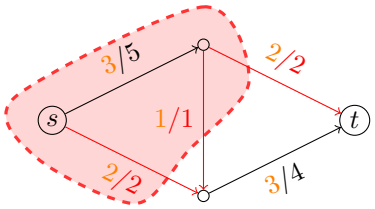
In the maximum flow problem we are given a network $\mathcal{N} = (V, A, c, s, t)$, which consists of a directed graph $D = (V, A)$ with capacities $c_e \geq 0$ for each $e \in A$ and two distinct vertices $s, t \in V$. The function $f : A \to \mathbb{R}$ is a flow if the following is satisfied:

- $0 \leq f(e) \leq c_e$ for all $e \in A$ (capacity constraint)
- $\sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} f(e)$ for all $v \in V \setminus \{s, t\}$ (flow conservation)

We note that the *value* of a feasible flow f is defined as $v(f) = \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$ and it is equal to $\sum_{e \in \delta^-(t)} f(e) - \sum_{e \in \delta^+(t)} f(e)$. The maximum flow problem is defined as follows.

Definition 1 (maximum flow problem)**.**
Input: A network $\mathcal{N} = (V, A, c, s, t)$.
Task: Compute an s-t flow in $\mathcal{N}$ of maximum value.

To compute such a maximum flow, we consider a closely related problem, called the Min s-t cut problem. For $U \subseteq V$ let $\delta^+(U) := \{(u, v) \in A \mid u \in U, v \in V \setminus U\}$. Let $U \subseteq V$ with $s \in U$ and $t \notin U$, then $C := \delta^+(U)$ is an s-t-Cut. The cut capacity is $\text{cap}(C) := \sum_{a \in C} c(a)$.



Definition 2 (minimum cut problem)**.**
Input: A network $\mathcal{N} = (V, A, c, s, t)$.
Task: Compute an s-t cut in $\mathcal{N}$ of minimum value.

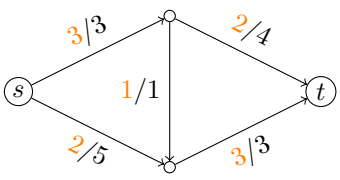
For a given flow f in some network, we define the **residual graph**, which helps us finding a maximum flow:

- Introduce backwards arc: $\overleftarrow{A} := \{\overleftarrow{a} \text{ such that } a \in A\}$
- Residual capacities for $a \in \overleftrightarrow{A} := A \cup \overleftarrow{A}$ are defined as

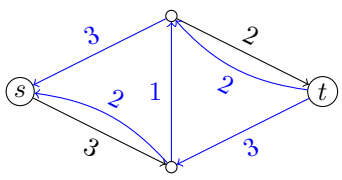
$$\bar{c}_f(a) := \begin{cases} c(a) - f(a) & \text{falls } a \in A \text{ (forward arc)} \\ f(a) & \text{falls } a \in \overleftarrow{A} \text{ (backward arc)} \end{cases}$$

  Arcs with $\bar{c}_f(a) = 0$ are deleted.

Then, the residual graph is $D_f = (V, A_f)$: $A_f := \{a \in \overleftrightarrow{A} \text{ such that } \bar{c}_f(a) > 0\}$.



Digraph D with flow f



Residual graph D_f

In order to increase a given flow f, we introduce augmenting paths, which are s-t paths P in the residual graph D_f. To increase the flow, we define the bottleneck capacity of P: $\gamma := \min_{a \in P} \bar{c}_f(a)$. Increasing the flow f along P by γ gives a flow f' in D:

$$f'(a) := \begin{cases} f(a) + \gamma & \text{if } a \in P \\ f(a) - \gamma & \text{if } \overleftarrow{a} \in P \\ f(a) & \text{otherwise.} \end{cases}$$

Using this, one can prove the following theorem.

Theorem 1. *f' ist a feasible s-t-flow in D with flow value $val(f') = val(f) + \gamma$.*

Furthermore, one can prove the following optimality criteria.

Theorem 2. *Let $N = (V, A, c, s, t)$ be a network and f a feasible s-t-flow. Then we have:*
f maximum $\Leftrightarrow \nexists$ f-augmenting s-t-path in the residual graph.

With small effort this theorem implies the following famous max-flow min-cut theorem.

Theorem 3. *Given a network $N = (V, A, c, s, t)$ with capacities $c(a) \geq 0$, $a \in A$. Then the value of a maximum s-t-flow is equal to the value of a minimum s-t-cut capacity.*

Hence, to find a flow of maximum value, one can iteratively search for augmenting paths in the residual graph and stop as soon as there is no augmenting path anymore. This is exactly what the Ford-Fulkerson Algorithm does.

Theorem 4. *If the arc capacities are integer, then the Ford-Fulkerson Algorithm computes an integer maximum s-t-flow in running-time $\mathcal{O}(m \cdot M)$, where M is the value of a maximum flow.*

Though this algorithm is correct, the running time is not as desired. Indeed, the running time is only *pseudo-polynomial* and hence in general can have an exponential running time (w.r.t. the input size). See also the exercise.

An improved algorithm is the Edmonds-Karp variant of the Ford-Fulkerson Algorithm: Do not just take an arbitrary augmenting path, but select a shortest one w.r.t. the number of arcs. This simple modification leads to a polynomial running time.

Theorem 5. *The variant of the Ford-Fulkerson Algorithm, in which the selected augmenting path is always a shortest s-t path in the residual graph (w.r.t. number of arcs), has a running time of $\mathcal{O}(m^2 \cdot n)$.*

Proof. To begin our analysis of the Edmonds-Karp Algorithm, note that the s–t path in D_f with the minimum number of edges can be found in $O(m)$ time using breadth-first search. (Generally, breadth-first search in a graph with n vertices and m edges requires $O(m+n)$ time, but our standing assumption that every vertex of the graph has at least one incident edge implies that $n \leq 2m$, from which it follows that $O(m+n) = O(m)$.) Once path P is discovered, it takes only $O(n)$ time to augment f using P and $O(n)$ time to update D_f, so — again using the fact that $n = O(m)$ — we see that one iteration of the while loop in the Edmonds-Karp Algorithm requires only $O(m)$ time. In the remainder, we bound the number of iterations.

To reason about the maximum number of while loop iterations, we take an indirect approach based on thinking about the breadth-first search tree of D_f, starting from s. (Henceforth we call this the *BFS tree* for short.) Recall that the vertices of the BFS tree can be organized into levels $L_0, L_1, \ldots, L_k$, where $L_0 = \{s\}$ and L_i ($i > 0$) consists of all the vertices v such that the path from s to v in the BFS tree has i edges. An elementary and useful property of BFS is the following: for all $v \in L_j$, every shortest path from s to v contains exactly one vertex from each of levels $L_0, L_1, \ldots, L_j$ (in that order) and no other vertices. In particular, every time the Edmonds-Karp algorithm chooses an augmenting path, that path consists of vertices $s = v_0, v_1, \ldots, v_j = t$ with $v_i \in L_i$ for $0 \leq i \leq j$.

Let us consider how the graph D_f changes when we augment f using P.

- If P contains a forward edge e, then edge e may be deleted from D_f (if the augmentation saturates e) and the backward edge $\overleftarrow{e}$ may be added to D_f (if D_f did not contain $\overleftarrow{e}$ before the augmentation).
- If P contains a backward edge $\overleftarrow{e}$, then $\overleftarrow{e}$ may be deleted from D_f (if the augmentation eliminates all flow on e) and the forward edge e may be added to D_f (if e had previously been saturated before the augmentation).

- No other edges are added or deleted.
- Thus, every new edge that is created when augmenting f using P is the reverse of an edge that belongs to P.

Recalling that every edge of P goes from level i to $i+1$, for some $0 \leq i < j$, we see that every new edge that gets created in D_f after the augmentation must go from level $i+1$ to level i, for some $0 \leq i < j$. In particular, for any vertex v, the distance from s to v never decreases as we run the Edmonds-Karp algorithm! (Creating edges that point from a higher-numbered level of the BFS tree to a lower-numbered level can never produce a "shortcut" that reduces the length of the shortest path from s to v.) This is the key property that guides our analysis of the algorithm.

When we choose the augmenting path P in D_f, let us say that edge $e \in E(D_f)$ is a *bottleneck edge* for P if $\bar{c}_f(e) = \gamma$. Notice that if $e = (u, v)$ is a bottleneck edge for P, then it is eliminated from D_f after augmenting f using P. Suppose that $u \in L_i$ and $v \in L_{i+1}$ when this happens. In order for e to be added back into D_f later on, u must occupy a higher-numbered level than v. (Recall that edges are only added to D_f when they point from one level to the immediately preceding level.) Since the distance from s to v never decreases, this means that v remains in level L_{i+1} or higher, and u must rise to level L_{i+2} or higher, before e is added back into D_f.

The BFS tree has no levels numbered above n. Thus, the total number of times that e can occur as a bottleneck edge during the Edmonds-Karp algorithm is at most $n/2$. There are $2m$ edges that can potentially appear in the residual graph, and each of them serves as a bottleneck edge at most $n/2$ times, so there are at most mn bottleneck edges in total. Every iteration of the while loop identifies an augmenting path, and that augmenting path must have a bottleneck edge, so there are at most mn while loop iterations in total. Since each iteration of the loop takes $O(m)$ time, the running time of the Edmonds-Karp Algorithm is $O(m^2 n)$.

□