Run time of Ford-Fulkerson: In worst-case, Ford-Fulkerson takes time $\Theta(mC)$, where C is sum of the capacity of all edges leaving the source s – this is not polytime.

Choosing augmenting paths efficiently:

- Edmonds-Karp algorithm: use BFS (modified to consider only augmenting edges) to find augmenting paths; guaranteed to find an augmenting path with smallest number of edges in time $O(V)$. Possible to prove that no more than $O(VE)$ augmentations are required to find max. flow. Total time: $O(VE^2) = O(V^5)$.
- Dinitz's algorithm: perform complete BFS, use all augmenting paths w.r.t. BFS tree, then repeat. Worst-case time down to $O(V^2E) = O(V^4)$.
- "Preflow-Push" algorithm: don't use augmenting paths; instead, push as much as possible along individual edges then go back to fix conservation. More complicated to explain and write down correctly, but cuts down time to $O(V^3)$.
- Why are we not concerned about details? Applications of network flow do not involve writing new algorithms: you always solve the same problem, so you can use an existing implementation that's already been debugged and optimized. Applications involve taking a problem, casting it in the guise of a network flow, solving the network flow problem, then casting the answer back to your problem.

Applications of network flows:

Multi-source, multi-sink network: Add "super-source" with edges of capacity ∞ to each source and "super-sink" with edges of capacity ∞ from each sink (instead of using ∞, we can set capacity to sum of outgoing/incoming capacities).
Max flow in resulting network = max flow in original network because:

- any flow in original network can be extended to a flow in resulting network (for new edges from super-source to source, set flow equal to total flow out of source; for new edges from sink to super-sink, set flow equal to total flow into sink) – hence, max flow in new network $\geqslant$ max flow in original network;
- any flow in resulting network induces flow in original network (flow out of every source and into every sink limited only by edges in original network because of "infinite" capacities on new edges) – hence, max flow in original network $\geqslant$ max flow in new network.

Maximum bipartite matching: We are given an undirected bipartite graph $G = (V_1, V_2, E)$ – one where every edge is between V_1 and V_2 (i.e., no edge has both endpoints in the same "side").
The goal is to identify a disjoint subset of edges of maximum size (i.e., no edge in a matching shares an endpoint with any other edge in the matching).

Given input graph, create network by turning every original edge into a directed edge (from V_1 to V_2) with capacity 1; add source with edges of capacity 1 to each vertex in V_1, sink with edges of capacity 1 from each vertex in V_2.

- Any matching in graph yields flow in network: set flow = 1 for graph edges in matching, 0 for graph edges not in matching; set flow equals 1 for new edges to/from matched vertices, 0 for new edges to/from unmatched vertices.
- Any integer flow in network yields matching in graph: pick edges with flow = 1 (leave out edges with flow = 0).

For this correspondence, size of matching = value of flow. Hence, any max flow in network yields a maximum matching in graph (because a larger matching would give a larger flow).

Difference between Network Flow and other techniques:
When solving a problem by "using network flows", what we are doing is actually a *reduction* or *transformation*: we take the input to our problem and create a network from it, then use standard algorithms to solve the maximum flow problem on that network (it's always the same problem and always the same algorithm, only the input differs). Then, we use the solution to the network flow problem to reconstruct a solution to our problem.

There are two ways that this could go wrong:

- The network we construct could fail to represent certain solutions to our original problem. We show this is *not* the case by arguing that every solution to the original problem yields a valid flow (or cut) in the network.
- The network we construct could have solutions that don't correspond to anything in our original problem. We show this is *not* the case by arguing that every flow (or, depending on the problem, every *integer* flow or every cut) in the network corresponds to a solution to the original problem.

Sometimes those arguments can be very short, because the correspondence is obvious between both problems. But both arguments are important and must always be included.

One last example:

Project selection: There exist a set of projects P each with revenue p_i (integer, can be positive or negative), and prerequisites between projects (given as directed graph on vertices P with edges (i, j) meaning project i depends on j). The goal is to find a *feasible* subset A of P with maximum total revenue (set A is feasible means that for each $i \in A$, A contains all i's prerequisites).

Unlike other examples, this can be solved by network flow techniques where we are interested in minimum cuts (flow values are irrelevant).

Construct network N from G as follows:

- Add source s, sink t,
- Add edges (s, i) with capacity p_i for each i such that $p_i \geqslant 0$,
- Add edges (i, t) with capacity $-p_i$ for each i such that $p_i < 0$,
- Set capacity of all other edges (i, j) to $C + 1$, where $C = \sum_{p_i>=0} p_i$.

Find a minimum cut (V_s, V_t) in N. Then $V_s - \{s\}$ is a feasible subset of projects with maximum profit! This requires proof...

Claim 1: for any cut (V_s, V_t), the set $V_s - \{s\}$ is feasible iff the capacity of the cut is at most C.
Proof: $c(V_s, V_t) \leqslant C$ implies no edge (i, j) (with capacity $C+1$) crosses the cut forward, i.e., for all projects $i \in V_s$, V_s contains all of i's prerequisites.

Claim 2: $c(A \cup \{s\}, \bar{A} \cup \{t\}) = C - \sum_{i\in A} p_i = C - profit(A)$ for all feasible sets of projects A (where $\bar{A} = P - A$).
Proof: Since A is feasible, no edge (i, j) with capacity $C + 1$ crosses the cut forward (some may cross backward). Forward edges crossing the cut fall into two groups:

- entering sink contribute $\sum_{i\in A, p_i<0} -p_i$
- leaving source contribute $\sum_{i\in\bar{A}, p_i\geqslant 0} p_i = C - \sum_{i\in A, p_i\geqslant 0} p_i$ (because $C = \sum_{p_i\geqslant 0} p_i = \sum_{i\in A, p_i\geqslant 0} p_i + \sum_{i\in\bar{A}, p_i\geqslant 0} p_i$)

So total capacity of cut is $\sum_{i\in A, p_i<0} -p_i + C - \sum_{i\in A, p_i\geqslant 0} p_i = C - \sum_{i\in A} p_i = C - profit(A)$.

These two facts imply that feasible sets of projects and cuts with capacity at most C correspond to each other. Since for any such set, $c(A \cup \{s\}, \bar{A} \cup \{t\}) = C - profit(A)$, and C is constant, any minimum-capacity cut (V_s, V_t) yields a maximum-profit set $V_s - \{s\}$.